Iliano Cervesato
Helmut Veith
Andrei Voronkov (Eds.)

LNAI 5330

Logic for Programming, Artificial Intelligence, and Reasoning

15th International Conference, LPAR 2008
Doha, Qatar, November 2008
Proceedings

Springer

Lecture Notes in Artificial Intelligence 5330

Edited by R. Goebel, J. Siekmann, and W. Wahlster

Subseries of Lecture Notes in Computer Science

Iliano Cervesato Helmut Veith
Andrei Voronkov (Eds.)

Logic for Programming, Artificial Intelligence, and Reasoning

15th International Conference, LPAR 2008
Doha, Qatar, November 22-27, 2008
Proceedings

Springer

Volume Editors

Iliano Cervesato
Carnegie Mellon University, Doha, Qatar

Helmut Veith
Technische Universität Darmstadt, Germany

Andrei Voronkov
University of Manchester, UK

Library of Congress Control Number: 2008939396

CR Subject Classification (1998): I.2.3, I.2, F.4.1, F.3, D.2.4, D.1.6

LNCS Sublibrary: SL 7 – Artificial Intelligence

ISSN 0302-9743
ISBN-10 3-540-89438-1 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-89438-4 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12573158 06/3180 5 4 3 2 1 0

Preface

This volume contains the papers presented at the 15th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR) held November 22–27 in Doha, Qatar on the premises of the Qatar campus of Carnegie Mellon University.

In its 15th edition, LPAR looked back at a rich history. The conference evolved out of the First and Second Russian Conferences on Logic Programming, held in Irkutsk, in 1990, and aboard the ship "Michail Lomonosov" in 1991. The idea of organizing the conference came largely from Robert Kowalski, who also proposed the creation of the Russian Association for Logic Programming. In 1992, it was decided to extend the scope of the conference. Due to considerable interest in automated reasoning in the former Soviet Union, the conference was renamed Logic Programming and Automated Reasoning (LPAR). Under this name three meetings were held during 1992–1994: again on board the ship "Michail Lomonosov" (1992), in St. Petersburg, Russia (1993), and on board the ship "Marshal Koshevoi" (1994). In 1999, the conference was held in Tbilisi, Georgia. At the suggestion of Michel Parigot, the conference changed its name again to Logic for Programming and Automated Reasoning (preserving the acronym LPAR!) reflecting an interest in additional areas of logic. LPAR 2000 was held on Reunion Island, France. In 2001, the name (but not the acronym) changed again to its current form. The 8th to the 14th meetings were held in the following locations: Havana, Cuba (2001) Tbilisi, Georgia (2002); Almaty, Kazakhstan (2003); Montevideo, Uruguay (2004); Montego Bay, Jamaica (2005); Phnom Penh, Cambodia (2006); and Yerevan, Armenia (2007).

Following the call for paper, LPAR 2008 received 207 abstracts which materialized into 154 actual submissions. Each submission was reviewed by at least three Program Committee members. Of these, the Program Committee selected 42 regular papers and 3 tool papers. The Committee deliberated electronically via the EasyChair system, which proved an egregious platform for smoothly carrying out all aspects of the program selection and finalization. It has been a tradition of LPAR to invite some of the most influential researchers in its focus area to discuss their work and their vision for the field. This year's distinguished speakers are Edmund Clarke (Carnegie Mellon University, USA—2007 Turing Award recipient), Amir Pnueli (New York University, USA—1996 Turing Award recipient), Michael Backes (Saarland University and MPI-SWS, Germany), and Thomas Eiter (Technical University of Vienna, Austria). This volume contains the revised versions of the 45 papers as well as the abstracts of the invited talks.

The conference was held in Doha, Qatar, where Carnegie Mellon University has recently established a branch campus with the goal of promoting the same high standards of research and education for which its original campus in Pittsburgh, USA is internationally recognized. Carnegie Mellon Qatar is located in

Education City, a 2,500 acre campus which provides state-of-the-art research and teaching facilities to branches of six of the world's leading universities. It is part of an unprecedented commitment of resources made by the Qatari leadership to position Qatar as a world-class center of education and research.

Many people have been involved in the organization of this conference. In particular, we wish to thank the Steering Committee for endorsing the candidacy of Doha for this year's edition of LPAR. This conference would not have been possible without the hard work of the many people who relentlessly handled the local arrangements, especially Thierry Sans and Kara Nesimiuk. We are most grateful to the 35 members of the Program Committee who did an excellent job in handling the large number of submissions, and the 202 additional reviewers who assisted them in their evaluations. We greatly appreciate the generous support of our sponsors, the Qatar National Research Fund (QNRF), the Qatar Science and Technology Park (QSTP), iHorizons, Carnegie Mellon University in Qatar, Microsoft Research and the Kurt Gödel Society. Finally we are grateful to the authors, the invited speakers and the attendees who made this conference an enjoyable and fruitful event.

October 2008

Iliano Cervesato
Helmut Veith
Andrei Voronkov

Organization

Steering Committee

Matthias Baaz	TU Vienna, Austria
Chris Fermüller	TU Vienna, Austria
Geoff Sutcliffe	University of Miami, USA
Andrei Voronkov	University of Manchester, UK

Program Chairs

Iliano Cervesato	Carnegie Mellon University, Qatar
Helmut Veith	Technical University of Darmstadt, Germany
Andrei Voronkov	University of Manchester, UK

Program Committee

Franz Baader	TU Dresden, Germany
Matthias Baaz	TU Vienna, Austria
Peter Baumgartner	National ICT, Australia
Josh Berdine	MSR Cambridge, UK
Armin Biere	Johannes Kepler University, Austria
Sagar Chaki	Carnegie Mellon SEI, USA
Hubert Comon-Lundh	ENS Cachan, France
Javier Esparza	TU Munich, Germany
Roberto Giacobazzi	University of Verona, Italy
Jürgen Giesl	RWTH Aachen, Germany
Orna Grumberg	Technion, Israel
Thomas Henzinger	EPFL, Switzerland
Joxan Jaffar	NUS, Singapore
Claude Kirchner	INRIA & LORIA, France
Stephan Kreutzer	Oxford University, UK
Orna Kupferman	Hebrew University, Israel
Alexander Leitsch	TU Vienna, Austria
Nicola Leone	University of Calabria, Italy
Heiko Mantel	TU Darmstadt, Germany
Catherine Meadows	Naval Research Laboratory, USA
Aart Middeldorp	University of Innsbruck, Austria
John Mitchell	Stanford University, USA
Andreas Podelski	University of Freiburg, Germany
Sanjiva Prasad	IIT Delhi, India
Alexander Razborov	Russian Academy of Sciences, Russia

Riccardo Rosati	University of Rome 1, Italy
Andrey Rybalchenko	MPI-SWS, Germany
Marko Samer	Durham University, UK
Ulrike Sattler	University of Manchester, UK
Torsten Schaub	University of Potsdam, Germany
Carsten Schuermann	IT University of Copenhagen, Denmark
Helmut Seidl	TU Munich, Germany
Henny Sipma	Stanford University, USA
Geoff Sutcliffe	University of Miami, USA
Ashish Tiwari	SRI, USA

Workshop Chair

Laura Kovács	EPFL, Switzerland

Publicity Chair

Geoff SutCliffe	University of Miami, USA

Local Organization

Iliano Cervesato	Carnegie Mellon University, Qatar
Thierry Sans	Carnegie Mellon University, Qatar

External Reviewers

Mario Alviano
Serge Autexier
Arnon Avron
Clemens Ballarin
Clark Barrett
Ulrich Berger
Piergiorgio Bertoli
Dietmar Berwanger
Gustavo Betarte
Meghyn Bienvenu
Frederic Blanqui
Manuel Bodirsky
Maria Bonet
Olivier Bournez
Paul Brauner
Paola Bruscoli
Tevfik Bultan
Guillaume Burel
James Caldwell
Francesco Calimeri
Krishnendu Chatterjee
Hubie Chen
Alessandro Cimatti
Evelyne Contejean
Nora Cuppens-Boulahia
Olivier Danvy
Anupam Datta
Cristina David
Jared C. Davis
Anuj Dawar
Jeremy Dawson
Stephanie Delaune

Nachum Dershowitz
Josee Desharnais
Joelle Despeyroux
Alessandra Di Pierro
Gilles Dowek
Luigi Dragone
Irène Durand
Steve Dworschak
Wolfgang Faber
David Faitelson
Stephan Falke
Oliver Fasching
Andrzej Filinski
Andrea Flexeder
Daniel Fridlender
Alexander Fuchs
Carsten Fuhs
Ulrich Furbach
Deepak Garg
Stephane Gaubert
Thomas Martin Gawlitza
Martin Gebser
Samir Genaim
Herman Geuvers
Hugo Gimbert
Bernhard Gramlich
Gianluigi Greco
Hans W. Guesgen
Florian Haftmann
James Harland
Ted Herman
Miki Hermann
Stefan Hetzl
Jochen Hoenicke
Markus Holzer
Matthew Horridge
Clement Houtmann
Florent Jacquemard
Arnav Jhala
Jean-Pierre Jouannaud
Lukasz Kaiser
Antonis C. Kakas
Deepak Kapur
Stefan Kiefer
Florent Kirchner
Pavel Klinov
Dmitry Korchemny
Konstantin Korovin
Alexander Krauss
Joerg Kreiker
Stephan Kreutzer
Viktor Kuncak
Oliver Kutz
Martin Lange
Hans Leiß
Domenico Lembo
Maurizio Lenzerini
Leonid Libkin
Dan Licata
Yang Liu
Michael Luttenberger
Carsten Lutz
Alexander Lux
Olivier Ly
Michael Maher
Johann Makowsky
Marco Maratea
Maarten Marx
Isabella Mastroeni
Richard Mayr
Yael Meller
Robert Mercer
John-Jules Meyer
Kevin Millikin
Niloy J. Mitra
Georg Moser
Ben Moszkowski
Rasmus E. Møgelberg
Koji Nakazawa
Juan Antonio Navarro Perez
Jorge Navas
Friedrich Neurauter
Hans de Nivelle
Michael Norrish
Hans Jürgen Ohlbach
Santiago Ontañón
Jens Otten
Martin Otto
Balaji Padmanabhan
Bijan Parsia

Etienne Payet
Yannick Pencole
Simona Perri
Michael Petter
Frank Pfenning
Paulo Pinheiro da Silva
Ruzica Piskac
Nir Piterman
Antonella Poggi
Marc Pouzet
Riccardo Puccella
Christophe Raffalli
Silvio Ranise
Francesco Ricca
Marco Roveri
Abhik Roychoudhury
Marco Ruzzi
Masahiko Sakai
Andrew Santosa
Jeffrey Sarnat
Abdul Sattar
Alexis Saurin
Francesco Scarcello
Sven Schewe
Peter Schneider-Kamp
Lutz Schröder
Norbert Schrimer
Dieter Schuster
Stefan Schwoon
Roberto Segala
Mohamed Nassim Seghir
Damien Sereni
Olha Shkaravska
Tijs van der Storm
Barbara Sprick
Lutz Strassburger
Ofer Strichman
Georg Struth
Heiner Stuckenschmidt
K. Subramani
Henning Sudbrock
Jun Sun
Bontawee Suntisrivaraporn
Dejvuth Suwimonteerabuth
Sven Thiele
Rene Thiemann
Peter Thiemann
Krishnaprasad Thirunarayan
Cesare Tinelli
Stephan Tobies
Tayssir Touili
Ralf Treinen
Dmitry Tsarkov
Christian Urban
Pawel Urzyczyn
K. Neeraj Verma
Rakesh Verma
Yakir Visel
Eelco Visser
Yakir Vizel
Razvan Voicu
Johannes Waldmann
Uwe Waldmann
Igor Walukiewicz
Volker Weber
Martin Wehrle
Stephanie Weirich
Daniel Weller
Bernd Westphal
Thomas Wilke
Verena Wolf
Frank Wolter
Bruno Woltzenlogel Paleo
Avi Yadgar
Greta Yorsh
G. Michael Youngblood
Harald Zankl
Yuanlin Zhang
Sarah Zobel

Table of Contents

Session 1. Constraint Solving

Session 2. Knowledge Representation 1

Session 3. Proof-Theory 1

Invited Talk

Session 4. Automata

Session 5. Linear Arithmetic

Session 6. Verification

Invited Talks

Session 7. Knowledge Representation 2

Session 8. Proof-Theory 2

Session 9. Quantified Constraints

Session 10. Modal and Temporal Logics

Session 11. Rewriting

Symmetry Breaking for Maximum Satisfiability*

Joao Marques-Silva[1], Inês Lynce[2], and Vasco Manquinho[2]

[1] School of Electronics and Computer Science, University of Southampton, UK
jpms@ecs.soton.ac.uk
[2] IST/INESC-ID, Technical University of Lisbon, Portugal
{ines,vmm}@sat.inesc-id.pt

Abstract. Symmetries are intrinsic to many combinatorial problems including Boolean Satisfiability (SAT) and Constraint Programming (CP). In SAT, the identification of symmetry breaking predicates (SBPs) is a well-known, often effective, technique for solving hard problems. The identification of SBPs in SAT has been the subject of significant improvements in recent years, resulting in more compact SBPs and more effective algorithms. The identification of SBPs has also been applied to pseudo-Boolean (PB) constraints, showing that symmetry breaking can also be an effective technique for PB constraints. This paper extends further the application of SBPs, and shows that SBPs can be identified and used in Maximum Satisfiability (MaxSAT), as well as in its most well-known variants, including partial MaxSAT, weighted MaxSAT and weighted partial MaxSAT. As with SAT and PB, symmetry breaking predicates for MaxSAT and variants are shown to be effective for a representative number of problem domains, allowing solving problem instances that current state of the art MaxSAT solvers could not otherwise solve.

1 Introduction

Symmetry breaking is a widely used technique for solving combinatorial problems. Symmetries have been extensively studied in Boolean Satisfiability (SAT) [15,4,7,1], and are regarded as an essential technique for solving specific classes of problem instances. Symmetries have also been widely used for solving constraint satisfaction problems (CSPs) [11]. More recent work has shown how to apply symmetry breaking in pseudo-Boolean (PB) constraints [2] and also in soft constraints [24]. It should be noted that symmetry breaking is viewed as an effective problem solving technique, either for SAT, PB or CP, that is often used as an optional technique, to be used when default algorithms are unable to solve a given problem instance.

In recent years there has been a growing interest in algorithms for MaxSAT and variants [16,17,26,13,14,18,21,20], in part because of the wide range of potential applications. MaxSAT and variants represent a more general framework than either SAT or PB, and so can naturally be used in many practical applications. The interest in MaxSAT and variants motivated the development of a new generation of MaxSAT algorithms, remarkably more efficient than early MaxSAT algorithms [25,5]. Despite the observed improvements, there are many problems still too complex for MaxSAT algorithms to

* This paper extends a preliminary technical report [19] on the same subject.

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 1–15, 2008.

solve [3]. Natural lines of research for improving MaxSAT algorithms include studying techniques known to be effective for either SAT, PB or CP. One concrete example is symmetry breaking. Despite its success in SAT, PB and CP, the usefulness of symmetry breaking for MaxSAT and variants has not been thoroughly investigated.

This paper addresses the problem of using symmetry breaking in MaxSAT and in its most well-known variants, partial MaxSAT, weighted MaxSAT and weighted partial MaxSAT. The work extends past recent work on computing symmetries for SAT [1] and PB constraints [2] by computing automorphisms on colored graphs obtained from CNF or PB formulas, and by showing how symmetry breaking predicates [7,1] can be exploited. The experimental results show that symmetry breaking is an effective technique for MaxSAT and variants, allowing solving problem instances that state of the art MaxSAT solvers could not otherwise solve.

The paper is organized as follows. The next section introduces the notation used throughout the paper, provides a brief overview of MaxSAT and variants, and also summarizes the work on symmetry breaking for SAT and PB constraints. Afterwards, the paper describes how to apply symmetry breaking in MaxSAT and variants. Experimental results, obtained on representative problem instances from the MaxSAT evaluation [3] and also from practical applications [1], demonstrate that symmetry breaking allows solving problem instances that could not be solved by *any* of the available state of the art MaxSAT solvers. The paper concludes by summarizing related work, by overviewing the main contributions, and by outlining directions for future work.

2 Preliminaries

This section introduces the notation used through the paper. Moreover, this section summarizes relevant results in symmetry identification and symmetry breaking, and develops extensions to existing results, which will serve for applying symmetry breaking in MaxSAT. Finally, this section also summarizes the MaxSAT problem and its variants.

2.1 Propositional Satisfiability

The usual definitions of propositional logic are assumed. Let $X = \{x_1, x_2, \ldots, x_n\}$ denote a set of propositional variables. A propositional formula φ in conjunctive normal form (CNF) is a conjunction of clauses. A clause ω is a disjunctions of literals. A literal is either a variable ($x \in X$) or its complement ($\bar{x}$, with $x \in X$). Where appropriate, clauses are viewed as sets of literals, defined on X, and CNF formulas are viewed as set of clauses.

A truth assignment is a function $\mathcal{A} : X \rightarrow \{0, 1\}$. The usual semantics of propositional logic is used for associating values with formulas given truth assignments to the variables. Assignments serve for computing the values of literals, clauses and the complete CNF formula, respectively, $\mathcal{A}(l)$, $\mathcal{A}(\omega)$ and $\mathcal{A}(\varphi)$ [1]. As a result, the truth value of literals, clauses and CNF formulas can be defined as follows:

[1] The use of $\mathcal{A}$ for describing the truth value of clauses and CNF formulas is an often used abuse of notation.

$$\mathcal{A}(l) = \begin{cases} \mathcal{A}(x_i) & \text{if } l = x_i \\ 1 - \mathcal{A}(x_i) & \text{if } l = \bar{x}_i \end{cases} \tag{1}$$

$$\mathcal{A}(\omega) = \max\{\mathcal{A}(l) \,|\, l \in \omega\} \tag{2}$$

$$\mathcal{A}(\varphi) = \min\{\mathcal{A}(\omega) \,|\, \omega \in \varphi\} \tag{3}$$

A clause is said to be *satisfied* if at least one of its literals assumes value 1. If all literals of a clause assume value 0, then the clause is *unsatisfied*. A formula is satisfied if all clauses are satisfied, otherwise it is unsatisfied. A truth assignment that satisfies φ is referred to as *model*. The set of models of φ is denoted by $\mathcal{M}(\varphi)$. The propositional satisfiability (SAT) problem consists in deciding whether there exists an assignment to the variables such that φ is satisfied.

2.2 Symmetries

A symmetry is an operation that preserves the constraints, and therefore also preserves the solutions of a problem instance [6]. For a set of symmetric objects, it is possible to obtain the whole set of objects from any of the objects. The elimination of symmetries has been extensively studied in CP and SAT [15,4,22,7]. With the goal of developing a solution for breaking symmetries in MaxSAT, this section provides a few necessary definitions related with symmetries in propositional formulas [7].

For a set X of variables, a *permutation* of X is a bijective function $\pi : X \to X$, and the image of x under π is denoted x^π. The set of all permutations of X is denoted by $\mathcal{P}_X$, and this set is a group under the composition operation. Permutations can be extended to literals, clauses and formulas, by replacing each literal by its permuted literal. As a result, $\varphi^\pi = \wedge_i \omega_i^\pi$, and $\omega_i^\pi = \vee_j l_i^\pi$. Moreover, $l_j^\pi = x_j^\pi$ if $l_j = x_j$, and $l_j^\pi = \overline{x_j^\pi}$ if $l_j = \bar{x}_j$.

Permutations also map truth assignments to truth assignments. If $\pi \in \mathcal{P}_X$, then each truth assignment $\mathcal{A}$ is mapped into a new truth assignment ${}^\pi\mathcal{A}$, where ${}^\pi\mathcal{A}(x) = \mathcal{A}(x^\pi)$.

Given a formula φ and $\pi \in \mathcal{P}_X$, π is a *symmetry* (or *automorphism*) iff $\varphi^\pi = \varphi$. Moreover, $\mathcal{S}_\varphi$ represents the set of symmetries of φ. A well-known result in symmetry breaking for SAT is the following [7]:

Proposition 1 (Proposition 2.1 in [7]). *Let φ be a CNF formula over X, $\pi \in \mathcal{S}_\varphi$, and $\mathcal{A}$ a truth assignment of X. Then $\mathcal{A} \in \mathcal{M}(\varphi)$ iff ${}^\pi\mathcal{A} \in \mathcal{M}(\varphi)$.*

Proposition 1 can be extended to account for the number of unsatisfied clauses given an assignment. Essentially, the number of unsatisfied clauses remains *unchanged* in the presence of permutations. A permutation maps each clause to another clause. For each assignment, each unsatisfied clause is mapped to another clause which is also unsatisfied.

Let $\mu(\varphi, \mathcal{A})$ denote the number of unsatisfied clauses of formula φ given assignment $\mathcal{A}$. Clearly $\mathcal{M}(\varphi) = \{\mathcal{A} \,|\, \mu(\varphi, \mathcal{A}) = 0\}$. Then the following holds:

Proposition 2. *Let φ be a CNF formula over X, $\pi \in \mathcal{S}_\varphi$, and $\mathcal{A}$ a truth assignment of X. Then $\mu(\varphi, \mathcal{A}) = \mu(\varphi^\pi, {}^\pi\mathcal{A})$.*

Proof: *The proof follows from the discussion above. Symmetries map clauses into clauses. Unsatisfied clauses will be mapped into unsatisfied clauses, and the mapping is one-to-one.* ■

Proposition 2 is used in the following sections for validating the correctness of symmetry breaking for MaxSAT and extensions.

It is also known that $\mathcal{S}_\varphi$ induces an equivalence relation on the truth assignments of X. Moreover, observe that for each equivalence class, the number of unsatisfied clauses is the same. Symmetry breaking predicates (SBPs) are used for selecting a reduced set of representatives from each equivalence class (ideally one representative from each equivalence class).

2.3 Symmetry Breaking

Given the definition of symmetries, symmetry breaking predicates target the elimination of all but one of the equivalent objects [7,1]. Symmetry breaking is expected to speed up the search as the search space gets reduced. For specific problems where symmetries may be easily found this reduction may be significant. Nonetheless, the elimination of symmetries necessarily introduces overhead that is expected to be negligible when compared with the benefits it may provide.

The most well-known strategy for eliminating symmetries in SAT consists in adding symmetry breaking predicates (SBPs) to the CNF formula [7]. SBPs are added to the formula before the search starts. The symmetries may be identified for each specific problem, and in that case it is required that the symmetries in the problem are identified when creating the encoding. Alternatively, one may give a formula to a specialized tool for detecting all the symmetries [1]. The resulting SBPs select one representative from each equivalence class. In case all symmetries are broken, only one assignment, instead of n assignments, may satisfy a set of constraints, n being the number of elements in a given equivalence class. The most often used approach for constructing SBPs consists in selecting the least assignment in each equivalence class, e.g. by implementing predicates that compare pairs of truth assignments. Other approaches include remodeling the problem [23] and breaking symmetries during search [12]. Remodeling the problem implies creating a different encoding, e.g. obtained by defining a different set of variables, in order to create a problem with less symmetries or even none at all. Alternatively, the search procedure may be adapted for adding SBPs as the search proceeds to ensure that any assignment symmetric to one assignment already considered will not be explored in the future, or by performing checks that symmetric equivalent assignments have not yet been visited.

Currently available tools for detecting and breaking symmetries for a given formula are based on group theory. From each formula a group is extracted, where a group is a set of permutations. A permutation is a one-to-one correspondence between a set and itself. Each symmetry defines a permutation on a set of literals. In practice, each permutation is represented by a product of disjoint cycles. Each cycle $(l_1\, l_2 \ldots l_m)$ with size m stands for the permutation that maps l_i on l_{i+1} (with $1 \leq i \leq m-1$) and l_m on l_1. Applying a permutation to a formula will produce exactly the same formula.

Example 1. Consider the following CNF formula:

$$\varphi = (x_1 \vee x_2) \wedge (\bar{x}_1 \vee x_2) \wedge (\bar{x}_2) \wedge (x_3 \vee x_2) \wedge (\bar{x}_3 \vee x_2)$$

The permutations identified for φ are $(x_3\ \bar{x}_3)$ and $(x_1\ x_3)(\bar{x}_1\ \bar{x}_3)$. (The permutation $(x_1\ \bar{x}_1)$ is implicit.) The formula resulting from the permutation $(x_3\ \bar{x}_3)$ is obtained by replacing every occurrence of x_3 by $\bar{x}_3$ and every occurrence of $\bar{x}_3$ by x_3. Clearly, the obtained formula is equal to the original formula. The same happens when applying the permutation $(x_1\ x_3)(\bar{x}_1\ \bar{x}_3)$: replacing x_1 by x_3, x_3 by x_1, $\bar{x}_1$ by $\bar{x}_3$ and $\bar{x}_3$ by $\bar{x}_1$ produces the same formula. From [7,1], selection of the least assignment in each permutation yields the symmetry breaking predicate $\varphi_{sbp} = (\bar{x}_3) \wedge (\bar{x}_1 \vee x_3)$.

2.4 Maximum Satisfiability

Given a propositional formula φ, the MaxSAT problem is defined as finding an assignment to variables in φ such that the number of satisfied clauses is maximized. (MaxSAT can also be defined as finding an assignment that minimizes the number of unsatisfied clauses.) Well-known variants of MaxSAT include partial MaxSAT, weighted MaxSAT and weighted partial MaxSAT.

For partial MaxSAT, a propositional formula φ is described by the conjunction of two CNF formulas φ_s and φ_h, where φ_s represents the *soft* clauses and φ_h represents the *hard* clauses. The partial MaxSAT problem over a propositional formula $\varphi = \varphi_h \wedge \varphi_s$ consists in finding an assignment to the problem variables such that all hard clauses (φ_h) are satisfied and the number of satisfied soft clauses (φ_s) is maximized.

For *weighted* MaxSAT, each clause in the CNF formula is associated to a non-negative weight. A weighted clause is a pair (ω, c) where ω is a classical clause and c is a natural number corresponding to the cost of unsatisfying ω. Given a weighted CNF formula φ, the *weighted* MaxSAT problem consists in finding an assignment to problem variables such that the total weight of the unsatisfied clauses is minimized, which implies that the total weight of the satisfied clauses is maximized.

For the *weighted partial* MaxSAT problem, the formula is the conjunction of a weighted CNF formula (soft clauses) and a classical CNF formula (hard clauses). The weighted partial MaxSAT problem consists in finding an assignment to the variables such that all hard clauses are satisfied and the total weight of satisfied soft clauses is maximized. Observe that, for both partial MaxSAT and weighted partial MaxSAT, hard clauses can also be represented as weighted clauses. For hard clauses one can consider that the weight is greater than the sum of the weights of the soft clauses. This allows a more uniform treatment of hard and weighted soft clauses.

MaxSAT and variants find a wide range of practical applications, that include scheduling, routing, bioinformatics, and design automation. Moreover, MaxSAT can be used for solving pseudo-Boolean optimization [14]. The practical applications of MaxSAT motivated recent interest in developing more efficient algorithms. The most efficient algorithms for MaxSAT and variants are based on branch and bound search, using dedicated bounding and inference techniques [16,17,13,14]. Lower bounding techniques include, for example, the use of unit propagation for identifying necessarily unsatisfied clauses, whereas inference techniques can be viewed as restricted forms of resolution, with the objective of simplifying the problem instance to solve.

3 Symmetry Breaking for MaxSAT

This section describes how to use symmetry breaking in MaxSAT. First, the construction process for the graph representing a CNF formula is briefly reviewed [7,1], as it will be modified later in this section. Afterwards, plain MaxSAT is considered. The next step is to address symmetry breaking for partial, weighted and weighted partial MaxSAT.

3.1 From CNF Formulas to Colored Graphs

Symmetry breaking for MaxSAT and variants requires a few modifications to the approach used for SAT [7,1]. This section summarizes the basic approach, which is then extended in the following sections.

Given a graph, the *graph automorphism* problem consists in finding isomorphic groups of edges and vertices with a one-to-one correspondence. In case of graphs with colored vertices, the correspondence is made between vertices with the same color. It is well-known that symmetries in SAT can be identified by reduction to a graph automorphism problem [7,1]. The propositional formula is represented as an undirected graph with colored vertices, such that the automorphism in the graph corresponds to a symmetry in the propositional formula.

Given a propositional formula φ, a colored undirected graph is created as follows:

- For each variable $x_j \in X$ add two vertices to represent x_j and $\bar{x}_j$. All vertices are associated with variables are colored with color 1;
- For each variable $x_j \in X$ add an edge between the vertices representing x_j and $\bar{x}_j$;
- For each binary clause $\omega_i = (l_j \vee l_k) \in \varphi$, add an edge between the vertices representing l_j and l_k;
- For each non-binary clause $\omega_i \in \varphi$ create a vertex colored with color 2;
- For each literal l_j in a non-binary clause ω_i, add an edge between the vertices representing the literal and the clause.

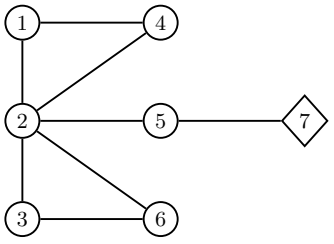
Example 2. Figure 1 shows the colored undirected graph associated with the CNF formula of Example 1. Vertices with shape $\circ$ represent color 1 and vertices with shape $\diamond$ represent color 2. Vertex 1 corresponds to x_1, 2 to x_2, 3 to x_3, 4 to $\bar{x}_1$, 5 to $\bar{x}_2$, 6 to $\bar{x}_3$ and 7 to unit clause $(\bar{x}_2)$. Edges 1-2, 2-3, 2-4 and 2-6 represent binary clauses and edges 1-4, 2-5 and 3-6 link complemented literals. Finally, edge 5-7 associates the correct literal with the unit clause.

Observe that for binary clauses it suffices to connect the vertices of the literals associated with the clause [1].

3.2 Plain Maximum Satisfiability

Let φ represent the CNF formula of a MaxSAT instance. Moreover, let φ_{sbp} be the CNF formula for the symmetry-breaking predicate obtained with a CNF symmetry tool (e.g. Shatter[2] [1] built on top of Saucy [8]). All clauses in φ are effectively *soft* clauses,

[2] Available from http://www.eecs.umich.edu/∼faloul/Tools/shatter/.



(a) Colored graph

Vertex	Mapping
1	literal x_1
2	literal $\bar{x}_1$
3	literal x_2
4	literal $\bar{x}_2$
5	literal x_3
6	literal $\bar{x}_3$
7	clause $(\bar{x}_2)$

(b) Vertex mapping

Fig. 1. Colored graph for Example 2

for which the objective is to maximize the number of satisfied clauses. In contrast, the clauses in φ_{sbp} are *hard* clauses, which must necessarily be satisfied. As a result, the original MaxSAT problem is transformed into a partial MaxSAT problem, where φ denotes the soft clauses and φ_{sbp} denotes the hard clauses. The solution of the partial MaxSAT problem corresponds to the solution of the original MaxSAT problem.

Example 3. As shown earlier, for the CNF formula of Example 1, the generated SBP (e.g. by Shatter) is: $\varphi_{sbp} = (\bar{x}_3) \land (\bar{x}_1 \lor x_3)$. As a result, the resulting instance of partial MaxSAT will be $\varphi' = (\varphi_h \land \varphi_s) = (\varphi_{sbp} \land \varphi)$. The addition of the clauses associated with the SBP imply $x_3 = 0$ and $x_1 = 0$. Observe that if there exists a MaxSAT solution for φ with $x_3 = 1$ or $x_1 = 1$, then not only it cannot have a smaller number of unsatisfied clauses than φ', but also such a solution must be included in an equivalent class for which there is at least one representative in the solutions of φ'.

As the previous example suggests, the hard clauses represented by φ_{sbp} do not change the solution of the original MaxSAT problem. Indeed, the construction of the symmetry breaking predicate guarantees that the maximum number of satisfied soft clauses remains unchanged by the addition of the hard clauses.

Proposition 3. *The maximum number of satisfied clauses for the MaxSAT problem φ and the partial MaxSAT problem $(\varphi \land \varphi_{sbp})$ are the same.*

Proof: *From Proposition 2 it is known that symmetries maintain the number of unsatisfied clauses, and this also holds for the equivalence classes induced by symmetries. Moreover, symmetry breaking predicates allow for at least one truth assignment from each equivalence class. Hence, at least one truth assignment from the equivalence class that maximizes the number of satisfied clauses will satisfy the symmetry breaking predicate, and so the solution of the MaxSAT problem is preserved.* ■

3.3 Partial and Weighted Maximum Satisfiability

For partial MaxSAT, the generation of SBPs needs to be modified. The graph representation of the CNF formula must take into account the existence of hard and soft clauses, which must be distinguished by a graph automorphism algorithm. Symmetric

objects for problem instances with hard and soft clauses establish a correspondence either between hard clauses or between soft clauses. In other words, when applying a permutation hard clauses can only be replaced by other hard clauses, and soft clauses by other soft clauses. In order to address this issue, the colored graph generation needs to be modified. In contrast to the MaxSAT case, binary clauses are *not* handled differently from other clauses, and must be represented as vertices in the colored graph. This is necessary for distinguishing between hard and soft binary clauses, and in general between binary clauses with different weights.

For the partial MaxSAT problem, clauses can now have one of two colors. A vertex with color 2 is associated with each soft clause, and a vertex with color 3 is associated with each hard clause. (As before, a vertex with color 1 corresponds to a literal.) This modification ensures that any identified automorphism guarantees that soft clauses correspond only to soft clauses, and hard clauses correspond only to hard clauses. Moreover, the procedure for the generation of SBPs from the groups found by a graph automorphism tool remains unchanged, and the SBPs can be added to the original instance as *new* hard clauses. The resulting instance is also an instance of partial MaxSAT. Correctness of this approach follows from the correctness of the plain MaxSAT case.

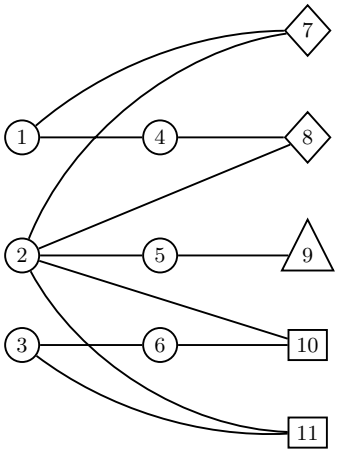
The solution for weighted MaxSAT and for weighted partial MaxSAT is similar to the partial MaxSAT case, but now clauses with different weights are represented by vertices with different colors. This guarantees that the groups found by the graph automorphism tool take into consideration the weight of each clause. Let $\{c_1, c_2, \ldots, c_k\}$ denote the distinct clause weights in the CNF formula. Each clause ω_i of weight c_i, represented as (ω_i, c_i) is associated with a vertex of color $i+1$ in the colored graph. In case there exist hard clauses, an additional color $k+2$ is used, and so each hard clause is represented by a vertex with color $k+2$ in the colored graph. Associating distinct clause weights with distinct colors guarantees that the graph automorphism algorithm can only make the correspondence between clauses with the same weight. Moreover, the identified SBPs result in new *hard* clauses that are added to the original problem. For either weighted MaxSAT or weighted partial MaxSAT, the result is an instance of weighted partial MaxSAT. As before, correctness of this approach follows from the correctness of the plain MaxSAT case.

Example 4. Consider the following weighted partial MaxSAT instance:

$$\varphi = (x_1 \vee x_2, 1) \wedge (\bar{x}_1 \vee x_2, 1) \wedge (\bar{x}_2, 5) \wedge (\bar{x}_3 \vee x_2, 9) \wedge (x_3 \vee x_2, 9)$$

for which the last two clauses are hard. Figure 2 shows the colored undirected graph associated with the formula. Clauses with different weights are represented with different colors (shown in the figure with different vertex shapes). A graph automorphism algorithm can then be used to generate the symmetry breaking predicates $\varphi_{sbp} = (\bar{x}_1) \wedge (\bar{x}_3)$, consisting of two hard clauses. As a result, the assignments $x_1 = 0$ and $x_3 = 0$ become necessary.

Proposition 4. *The maximum number of satisfied clauses for the weighted (partial) MaxSAT problem φ and the resulting weighted partial MaxSAT problem $(\varphi \wedge \varphi_{sbp})$ are the same.*



(a) Colored graph

Vertex	Mapping
1	literal x_1
2	literal x_2
3	literal x_3
4	literal $\bar{x}_1$
5	literal $\bar{x}_2$
6	literal $\bar{x}_3$
7	clause $(x_1 \vee x_2, 1)$
8	clause $(\bar{x}_1 \vee x_2, 1)$
9	clause $(\bar{x}_2, 5)$
10	clause $(\bar{x}_3 \vee x_2, 9)$
11	clause $(x_3 \vee x_2, 9)$

(b) Vertex mapping

Fig. 2. Colored graph for Example 2

Table 1. Problem transformations due to SBPs

Original	MS	PMS	WMS	WPMS
With Symmetries	PMS	PMS	WPMS	WPMS

Proof: *(Sketch) The proof is similar to the proof of Proposition 3, but noting that weights partition the set of clauses into sets of clauses that can be mapped into each other. Since mappings are between clauses with the same weights, the previous results (from Propositions 2 and 3) still hold.* ■

Table 1 summarizes the problem transformations described in this section, where MS represents plain MaxSAT, PMS represents partial MaxSAT, WMS represents weighted MaxSAT, and WPMS represents weighted partial MaxSAT. The use of SBPs introduces a number of hard clauses, and so the resulting problems are either partial MaxSAT or weighted partial MaxSAT.

3.4 Evaluating Alternative Formulations

Even though the proposed approach for breaking symmetries does not seem amenable to further optimizations for the MaxSAT and partial MaxSAT cases, it is interesting to investigate whether it is possible to optimize the approach outlined in the previous section for the weighted variants of MaxSAT, e.g. by reorganizing clause weights. This section argues that, provided some simple conditions hold, rearranging weights cannot induce stronger symmetry breaking predicates.

Example 5. Consider the weighted MaxSAT formula:

$$\varphi = (x_1 \vee \bar{x}_2, 7) \wedge (\bar{x}_3 \vee x_4, 3) \wedge (\bar{x}_3 \vee x_4, 4) \tag{4}$$

The symmetries for this formula are $(x_1\,\bar{x}_2)$ and $(x_3\,\bar{x}_4)$. Clearly, it is possible to induce more symmetries by considering the following modification:

$$\varphi = (x_1 \vee \bar{x}_2, 3) \wedge (x_1 \vee \bar{x}_2, 4) \wedge (\bar{x}_3 \vee x_4, 3) \wedge (\bar{x}_3 \vee x_4, 4) \tag{5}$$

In addition to the previous symmetries, one now also obtains $(x_1\,x_3)(x_2\,x_4)$.

The previous example suggests that by rearranging weights one may be able to increase the number of identified symmetries. As the example also suggests, this can only happen when a clause is associated with *more* than one single weight. For the previous example $(\bar{x}_3 \vee x_4)$ is associated with weights 3 and 4. One simple way to tackle this problem is to require that multiple occurrences of the same clause be aggregated into a single clause, i.e. multiple occurrences of the same clause are represented by a single clause and the multiple weights are added.

Proposition 5. *If each clause has a single occurrence in formula φ, then splitting the weight of a clause induces no additional symmetries.*

Proof: *Suppose that each clause has a single occurrence, and that additional symmetries could be identified by splitting the weight c_i of a single clause ω_i. Without loss of generality assume that weight c_i is split into c_{i_1} and c_{i_2}. If additional symmetries can now be identified, then ω_i is mapped to clause ω_{j_1} due to c_{i_1} and to clause ω_{j_2} due to c_{i_2}. However, since each variable is mapped to some other variable, then ω_{j_1} and ω_{j_2} must be the same clause; but this is a contradiction.* ■

The previous result ensures that the approach outlined in Section 3.3, for computing symmetry breaking predicates for the weighted variations of MaxSAT, cannot be improved upon by rearranging clause weights, provided each clause has a single occurrence in the formula. Clearly, this is not the case with the earlier example.

4 Experimental Results

The approach outlined in the previous sections for generating SBPs for MaxSAT has been implemented in MAXSATSBP [3]. MAXSATSBP interfaces SAUCY [8], and is organized similarly to SHATTER [1] and SHATTERPB [2].

The experimental setup has been organized as follows. First, all the instances from the first and second MaxSAT evaluations (2006 and 2007) [3] were run. A timeout of 1000s of CPU time was considered, and instances requiring more than 1000s of CPU time are declared as *aborted*. These results allowed selecting relevant benchmark families, for which symmetries occur and which require a non-negligible amount of time for being solved by both approaches (with or without SBPs). Afterwards, the instances for which both approaches aborted were removed from the tables of results. This resulted in selecting the `hamming` and the `MANN` instances for plain MaxSAT, the `ii32` and again the `MANN` instances for partial MaxSAT, the `c-fat500` instances for weighted MaxSAT and the `dir` and `log` instances for weighted partial MaxSAT.

[3] The MAXSATSBP tool is available on request from the authors.

Besides the instances that participated in the MaxSAT competition, we have included additional plain MaxSAT problem instances (`hole`, `Urq` and `chnl`). The `hole` instances refer to the well-known pigeon hole problem, the `Urq` instances represent randomized instances based on expander graphs and the `chnl` instances model the routing of wires in the channels of field-programmable integrated circuits. These instances refer to problems that can be naturally encoded as MaxSAT problems and are known to be highly symmetric [1]. The approach outlined above was also followed for selecting the instances to be included in the results.

We have run different publicly available MaxSAT solvers, namely MINIMAXSAT [4], TOOLBAR [5] and MAXSATZ [6]. (MAXSATZ accepts only plain MaxSAT instances.) Evidence from the MaxSAT evaluation suggests that the behavior of MINIMAXSAT is similar to TOOLBAR and MAXSATZ, albeit being in general more robust. For this reason, the results focus on MINIMAXSAT.

Tables 2 and 3 provide the results obtained. In the tables, TO denotes a timeout, and so the run time is in excess of 1000s. Table 2 refers to plain MaxSAT instances and Table 3 refers to partial MaxSAT (PMS), weighted MaxSAT (WMS) and weighted partial MaxSAT (WPMS) instances. For each instance, the results shown include the number of clauses added as a result of SBPs (#ClsSbp), the time required for solving the original instances (OrigT), i.e. without SBPs, and the time required for breaking the symmetries plus the time required for solving the extended formula afterwards (SbpT). (The best configuration for each instance is outlined in bold.) Moreover, the SbpT column is split into the time to run MAXSATSBP (MXSBP) and the time to run MINIMAXSAT. In practice, the time required for generating SBPs is negligible. The results were obtained on an Intel Xeon 5160 server (3.0GHz, 1333Mhz FSB, 4MB cache) running Red Hat Enterprise Linux WS 4.

The experimental results allow establishing the following conclusions:

- The inclusion of symmetry breaking is *essential* for solving a number of problem instances. We should note that *all* the plain MaxSAT instances in Table 2 for which MINIMAXSAT aborted, are also aborted by TOOLBAR and MAXSATZ. After adding SBPs all these instances become easy to solve by any of the solvers. For the aborted partial, weighted and weighted partial MaxSAT instances in Table 3 this is not always the case, since a few instances aborted by MINIMAXSAT could be solved by TOOLBAR without SBPs. However, the converse is also true, as there are instances that were initially aborted by TOOLBAR (although solved by MINIMAXSAT) that are solved by TOOLBAR after adding SBPs.
- For several instances, breaking only a few symmetries can make the difference. We have observed that in some cases the symmetries are broken with unit clauses.
- Adding SBPs is beneficial for most cases where symmetries exist. However, for a few examples, SBPs may degrade performance.
- There is no clear relation between the number of SBPs added and the impact on the search time.
- The run time of the symmetry breaking tool is in general negligible.

[4] http://www.lsi.upc.edu/∼fheras/docs/m.tar.gz
[5] http://carlit.toulouse.inra.fr/cgi-bin/awki.cgi/ToolBarIntro
[6] http://www.laria.u-picardie.fr/∼cli/maxsatz.tar.gz

Table 2. Results for MINIMAXSAT on plain MaxSAT instances

Name	#ClsSbp	OrigT	SbpT	MXSBP	MiniMaxSat
hamming10-2	81	TO	**0.19**	0.009	0.178
hamming10-4	1	886.57	**496.79**	0.01	496.777
hamming6-4	437	0.17	**0.15**	0.013	0.137
hamming8-2	85	TO	**0.21**	0.016	0.189
hamming8-4	253	0.36	**0.11**	0.011	0.102
MANN_a27	85	TO	**0.24**	0.012	0.226
MANN_a45	79	TO	**0.20**	0.011	0.185
MANN_a81	79	TO	**0.19**	0.01	0.184
hole10	758	42.11	**0.24**	0.023	0.213
hole11	922	510.90	**0.47**	0.023	0.442
hole12	1102	TO	**1.78**	0.028	1.752
hole7	362	**0.10**	0.11	0.007	0.103
hole8	478	0.40	**0.13**	0.008	0.122
hole9	610	3.68	**0.17**	0.016	0.15
Urq3_5	29	83.33	**0.27**	0.033	0.236
Urq4_5	43	TO	**50.88**	0.07	50.806
chnl10_11	1954	TO	**41.79**	0.053	41.737
chnl10_12	2142	TO	**328.12**	0.057	328.063
chnl11_12	2370	TO	**420.19**	0.075	420.111

Table 3. Results for MINIMAXSAT on partial, weighted and weighted partial MaxSAT instances

Name	MStype	#ClsSbp	OrigT	SbpT	MXSBP	MiniMaxSat
ii32e3	PMS	1756	94.40	**37.63**	0.482	37.15
ii32e4	PMS	2060	175.07	**129.06**	0.787	128.277
c-fat500-10	WMS	2	57.79	**11.62**	0.028	11.591
c-fat500-1	WMS	112	**0.03**	0.06	0.016	0.046
c-fat500-2	WMS	12	0.16	**0.11**	0.011	0.049
c-fat500-5	WMS	4	0.16	**0.11**	0.016	0.091
MANN_a27	WMS	1	TO	**880.58**	0.047	880.533
MANN_a45	WMS	1	TO	**530.86**	0.048	530.807
MANN_a81	WMS	1	TO	**649.13**	0.042	649.084
1502.dir	WPMS	1560	**0.34**	10.67	0.754	9.912
29.dir	WPMS	132	TO	**28.09**	0.031	28.055
54.dir	WPMS	98	4.14	**0.32**	0.029	0.292
8.dir	WPMS	58	**0.03**	0.05	0.008	0.039
1502.log	WPMS	812	0.76	**0.71**	0.32	0.385
29.log	WPMS	54	17.55	**0.82**	0.026	0.792
404.log	WPMS	124	TO	**64.24**	0.094	64.151
54.log	WPMS	48	2.37	**0.16**	0.021	0.139

Overall, the inclusion of SBPs should be considered when a hard problem instance is known to exhibit symmetries. This does not necessarily imply that after breaking

symmetries the instance becomes trivial to solve, and there can be cases where the new clauses may degrade performance. However, in a significant number of cases, highly symmetric problems become much easier to solve after adding SBPs. In many of these cases the problem instances become *trivial* to solve.

5 Related Work

Symmetries are a well-known research topic, that serve to tackle complexity in many combinatorial problems. The first ideas on symmetry breaking were developed in the 80s and 90s [15,4,22,7], by relating symmetries with the graph automorphism problem, and by proposing the first approach for generating symmetry breaking predicates. This work was later extended and optimized for propositional satisfiability [1].

Symmetries are an active research topic in CP [11]. Approaches for breaking symmetries include not only adding constraints before search [22] but also reformulation [23] and dynamic symmetry breaking methods [12]. Recent work has also shown the application of symmetries to soft CSPs [24].

The approach proposed in this paper for using symmetry breaking for MaxSAT and variants builds on earlier work on symmetry breaking for PB constraints [2]. Similarly to the work for PB constraints, symmetries are identified by constructing a colored graph, from which graph automorphisms are obtained, which are then used to generate the symmetry breaking predicates.

6 Conclusions

This paper shows how symmetry breaking can be used in MaxSAT and in its most well-known variants, including partial MaxSAT, weighted MaxSAT, and weighted partial MaxSAT. Experimental results, obtained on representative instances from the MaxSAT evaluation [3] and practical instances [1], demonstrate that symmetry breaking allows solving problem instances that no state of the art MaxSAT solver could otherwise solve. For all problem instances considered, the computational effort of computing symmetries is negligible. Nevertheless, and as it is the case with symmetry breaking for SAT and PB constraints, symmetry breaking should be considered as an optional problem solving technique, to be used when standard techniques are unable to solve a given problem instance.

The experimental results motivate additional work on computing symmetry breaking predicates for MaxSAT. A new more efficient version of Saucy has recently been developed [9] and is likely to further reduce the run time for computing symmetries. Moreover, the use of conditional symmetries could be considered [10,24].

Acknowledgement. This work is partially supported by EPSRC grant EP/E012973/1, by EU grants IST/033709 and ICT/217069, and by FCT grants POSC/EIA/61852/2004 and PTDC/EIA/76572/2006.

References

1. Aloul, F., Sakallah, K.A., Markov, I.: Efficient symmetry breaking for boolean satisfiability. In: International Joint Conference on Artificial Intelligence, pp. 271–276 (August 2003)
2. Aloul, F.A., Ramani, A., Markov, I.L., Sakallah, K.A.: ShatterPB: symmetry-breaking for pseudo-Boolean formulas. In: Asian and South-Pacific Design Automation Conference, pp. 883–886 (2004)
3. Argelich, J., Li, C.M., Manyà, F., Planes, J.: MaxSAT evaluation (May 2007), www.maxsat07.udl.es
4. Benhamou, B., Sais, L.: Theoretical study of symmetries in propositional calculus and applications. In: Eleventh International Conference on Automated Deduction, pp. 281–294 (1992)
5. Borchers, B., Furman, J.: A two-phase exact algorithm for MAX-SAT and weighted MAX-SAT problems. Journal of Combinatorial Optimization 2(4), 299–306 (1998)
6. Cohen, D.A., Jeavons, P., Jefferson, C., Petrie, K.E., Smith, B.M.: Symmetry definitions for constraint satisfaction problems. In: International Conference on Principles and Practice of Constraint Programming, pp. 17–31 (2005)
7. Crawford, J.M., Ginsberg, M.L., Luks, E., Roy, A.: Symmetry-breaking predicates for search problems. In: International Conference on Principles of Knowledge Representation and Reasoning, pp. 148–159 (1996)
8. Darga, P.T., Liffiton, M.H., Sakallah, K.A., Markov, I.L.: Exploiting structure in symmetry detection for CNF. In: Design Automation Conference, pp. 530–534 (June 2004)
9. Darga, P.T., Sakallah, K.A., Markov, I.L.: Faster symmetry discovery using sparsity of symmetries. In: Design Automation Conference, pp. 149–154 (June 2008)
10. Gent, I.P., Kelsey, T., Linton, S., McDonald, I., Miguel, I., Smith, B.M.: Conditional symmetry breaking. In: International Conference on Principles and Practice of Constraint Programming, pp. 256–270 (2005)
11. Gent, I.P., Petrie, K.E., Puget, J.-F.: Symmetry in Constraint Programming. In: Handbook of Constraint Programming, pp. 329–376. Elsevier, Amsterdam (2006)
12. Gent, I.P., Smith, B.M.: Symmetry breaking in constraint programming. In: European Conference on Artificial Intelligence, pp. 599–603 (2000)
13. Heras, F., Larrosa, J.: New inference rules for efficient Max-SAT solving. In: AAAI Conference on Artificial Intelligence (2006)
14. Heras, F., Larrosa, J., Oliveras, A.: MiniMaxSat: a new weighted Max-SAT solver. In: International Conference on Theory and Applications of Satisfiability Testing, pp. 41–55 (May 2007)
15. Krishnamurthy, B.: Short proofs for tricky formulas. Acta Informatica 22(3), 253–275 (1985)
16. Li, C.M., Manyà, F., Planes, J.: Exploiting unit propagation to compute lower bounds in branch and bound Max-SAT solvers. In: International Conference on Principles and Practice of Constraint Programming, pp. 403–414 (2005)
17. Li, C.M., Manyà, F., Planes, J.: Detecting disjoint inconsistent subformulas for computing lower bounds for Max-SAT. In: AAAI Conference on Artificial Intelligence (July 2006)
18. Lin, H., Su, K.: Exploiting inference rules to compute lower bounds for MAX-SAT solving. In: International Joint Conference on Artificial Intelligence, pp. 2334–2339 (2007)
19. Marques-Silva, J., Lynce, I., Manquinho, V.: Symmetry breaking for maximum satisfiability. Computing Research Repository, abs/0804.0599 (April 2008), http://arxiv.org/abs/0804.0599
20. Marques-Silva, J., Manquinho, V.M.: Towards more effective unsatisfiability-based maximum satisfiability algorithms. In: International Conference on Theory and Applications of Satisfiability Testing, pp. 225–230 (2008)

21. Marques-Silva, J., Planes, J.: Algorithms for maximum satisfiability using unsatisfiable cores. In: Design, Automation and Testing in Europe Conference, pp. 408–413 (March 2008)
22. Puget, J.-F.: On the satisfiability of symmetrical constrained satisfaction problems. In: International Symposium on Methodologies for Intelligent Systems, pp. 350–361 (1993)
23. Smith, B.M.: Reducing symmetry in a combinatorial design problem. Technical Report 2001.01, School of Computing, University of Leeds (January 2001) (Presented at the CP-AI-OR Workshop April 2001)
24. Smith, B.M., Bistarelli, S., O'Sullivan, B.: Constraint symmetry for the soft CSP. In: International Conference on Principles and Practice of Constraint Programming, pp. 872–879 (September 2007)
25. Wallace, R., Freuder, E.: Comparative studies of constraint satisfaction and Davis-Putnam algorithms for maximum satisfiability problems. In: Johnson, D., Trick, M. (eds.) Cliques, Coloring and Satisfiability. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 26, pp. 587–615. American Mathematical Society (1996)
26. Xing, Z., Zhang, W.: MaxSolver: An efficient exact algorithm for (weighted) maximum satisfiability. Artificial Intelligence 164(1-2), 47–80 (2005)

Efficient Generation of Unsatisfiability Proofs and Cores in SAT

Roberto Asín, Robert Nieuwenhuis, Albert Oliveras,
and Enric Rodríguez-Carbonell*

Abstract. Some modern DPLL-based propositional SAT solvers now have fast in-memory algorithms for generating unsatisfiability proofs and cores without writing traces to disk. However, in long SAT runs these algorithms still run out of memory.

For several of these algorithms, here we discuss advantages and disadvantages, based on carefully designed experiments with our implementation of each one of them, as well as with (our implementation of) Zhang and Malik's one writing traces on disk. Then we describe a new in-memory algorithm which saves space by doing more bookkeeping to discard unnecessary information, and show that it can handle significantly more instances than the previously existing algorithms, at a negligible expense in time.

1 Introduction

More and more applications of propositional SAT solvers and their extensions keep emerging. For some of these applications, it suffices to obtain a yes/no answer, possibly with a model in case of satisfiability. For other applications, also in case of *un*satisfiability a more detailed answer is needed. For example, one may want to obtain a small (or even minimal, wrt. set inclusion) unsatisfiable subset of the initial set of clauses. Such subsets, called *unsatisfiable cores*, are obviously useful in applications like planning or routing for explaining why no feasible solution exists, but many other applications keep emerging, such as solving MAX-SAT problems [FM06, MSP08] or debugging software models [Jac02].

In addition, it is frequently helpful, or even necessary, to be able to check the unsatisfiability claims produced by a DPLL-based ([DP60, DLL62]) SAT solver, using some small and simple, independent, trusted checker for, e.g., resolution proofs. Note that, although for certain classes of formulas the minimal resolution proof is exponentially large [Hak85], for real-world problems the size tends to be manageable and frequently it is in fact surprisingly small (as well as the core).

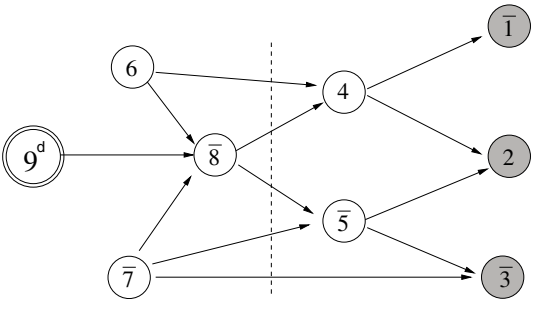
Since Zhang and Malik's work in 2003 [ZM03], it is well-known that modern DPLL-based solvers with learning can be instrumented to write a trace on disk from which a resolution proof can be extracted and checked. Essentially, each learned clause generates a line in the trace with only the list of its parents' identifiers (ID's), i.e., the ID's of the clauses involved in the conflict analysis, which

* Technical Univ. of Catalonia, Barcelona. Partially supported by Spanish Ministry Educ. and Science LogicTools-2 project (TIN2007-68093-C02-01).

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 16–30, 2008.

is a sequence of resolution steps (see the example below). When unsatisfiability is detected, that is, a conflict at decision level zero, it provides a last line in the trace corresponding to the parents list of the empty clause. By processing the trace file backwards from this last line one can hence reconstruct a resolution proof and find the subset of the initial clauses that is used in it.

Example 1. (see Section 2 for details) Consider, among others, a set of clauses:
$\overline{9}\vee\overline{6}\vee 7\vee\overline{8}$ $\quad 8\vee 7\vee\overline{5}$ $\quad \overline{6}\vee 8\vee 4$ $\quad \overline{4}\vee\overline{1}$ $\quad \overline{4}\vee 5\vee 2$ $\quad 5\vee 7\vee\overline{3}$ $\quad 1\vee\overline{2}\vee 3$
and a state of the DPLL procedure where the stack of assigned literals is of the form: $\ldots 6\ldots\overline{7}\ldots 9^{\mathsf{d}}\,\overline{8}\,\overline{5}\,4\,\overline{1}\,2\,\overline{3}$. It is easy to see that this state can be reached after the last *decision* 9^{d} by six *unit propagation* steps with these clauses (from left to right). For example, $\overline{8}$ is implied by 9, 6, and $\overline{7}$ because of the leftmost clause. Now, the clause $1\vee\overline{2}\vee 3$ is *conflicting* (it is false in the current assignment), and working backwards from it we get an *implication graph*:



where the so-called *1UIP cut* (the dotted line, see [MSS99, MMZ+01]) gives us the *backjump clause* $8\vee 7\vee\overline{6}$ that is *learned* as a lemma. For those who are more familiar with resolution, this is simply a backwards resolution proof on the conflicting clause, resolving away the literals $3, \overline{2}, 1, \overline{4}$ and 5, in the reverse order their negations were propagated, with the respective clauses that caused the propagations:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{5\vee 7\vee\overline{\mathbf{3}} \qquad 1\vee\overline{2}\vee\mathbf{3}}{\overline{4}\vee 5\vee\mathbf{2} \qquad 5\vee 7\vee 1\vee\overline{\mathbf{2}}}}{\overline{4}\vee\overline{\mathbf{1}} \qquad \overline{4}\vee 5\vee 7\vee\mathbf{1}}}{\overline{6}\vee 8\vee\mathbf{4} \qquad 5\vee 7\vee\overline{\mathbf{4}}}}{8\vee 7\vee\overline{\mathbf{5}} \qquad \overline{6}\vee 8\vee 7\vee\mathbf{5}}}{8\vee 7\vee\overline{6}}$$

until reaching a clause with only one literal of the current decision level (here, literal 8). This clause $8\vee 7\vee\overline{6}$ allows one to *backjump* to the state $\ldots 6\ldots\overline{7}\,8$, as if it had been used on $\ldots 6\ldots\overline{7}$ for unit propagation.

Due to the linear and regular nature of such resolution proofs (each literal is resolved upon at most once and then does not re-appear), given the six input clauses it is easy to see that the outcome must be $8\vee 7\vee\overline{6}$: its literals are exactly the ones that do not occur with both polarities in the input clauses. This fact allows one to reconstruct and check the whole resolution proof from (i) the input clauses file and (ii) the parent ID information of the trace file. Note that a clause's ID can just be the line number (in one of the files) introducing it. □

The overhead in time for producing the trace file is usually small (typically around 10 per cent, [ZM03], see Section 4), but the traces quickly become large (hundreds of MB from a few minutes run) and extracting the proof or the core, i.e., the leaves of the proof DAG, may be expensive. This is especially the case since it is likely that the trace does not fit into memory, and hence a breadth-first processing is needed requiring more than one pass over the file [ZM03].

The processing time becomes even more important when, for reducing the size of the core, one iteratively feeds it back into the SAT solver with the hope of generating a smaller one, until a fixpoint is reached (that still may not be minimal, so one can apply other methods for further reducing it, if desired). Efficiency is also important in other applications requiring features like the identification of all disjoint cores, i.e., all independent reasons for unsatisfiability.

1.1 In-Memory Algorithms

To overcome the drawbacks of the trace file approach, in this paper we study four alternative in-memory algorithms for generating unsatisfiability proofs and cores using DPLL-based propositional SAT solvers.

The first algorithm is based on adding one distinct new *initial ancestor (IA)* marker literal to each initial clause. These literals are set to false from the beginning. Then the solver is run without ever removing these false IA-marker literals from clauses, and the empty clause manifests itself as a clause built solely from IA-marker literals, each one of which identifies one initial ancestor, that is, one clause of the unsatisfiable core. This *folk* idea appears to be quite widely applied (e.g., in SAT Modulo Theories). As far as we know, it stems from the Minisat group (around 2002, Eén, Sörensson, Claessen). It requires little implementation effort, but here, in Subsection 3.1 we give experimental evidence showing that it is extremely inefficient in solver time and memory and explain why.

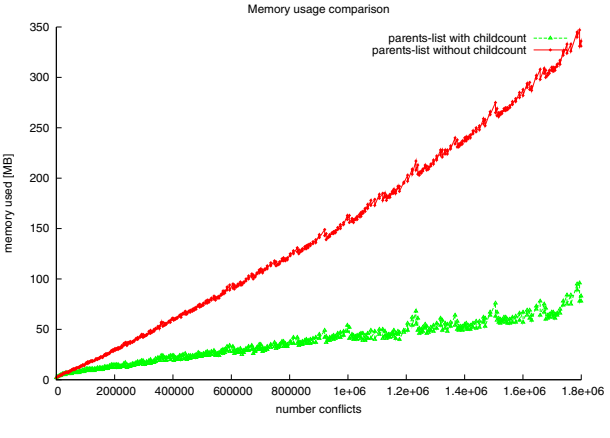
Our second algorithm, given in Subsection 3.2, tries to overcome these shortcomings by storing initial ancestor information at the meta level along with the clauses: each clause has an attached list with the ID's of its initial ancestors. This reduces part of the overhead of the first algorithm. However, our experiments reveal that also this method is still far too expensive in memory, especially in combination with certain clause simplification methods, which on the other hand, when turned off, slow down the solver too much.

The third algorithm (Section 4.1) stores the immediate parents list along with each clause. The problem with this approach is that if a low-activity clause is deleted (as usual in modern SAT solvers), its associated parent information can be removed only if this clause has generated no children (the *literals* of deleted clauses need not be kept, though). This approach, implemented by Biere in PicoSAT [Bie08], essentially corresponds to storing the trace file of [ZM03] in main memory. In those cases where this is indeed feasible, i.e., if there is enough memory, this has several advantages over the trace file one. One not only avoids the inefficiencies caused by the use of external memory, but also, and more importantly, for retrieving the proof or the core one does not need to sequentially traverse the *whole* trace, but only those parts of it that appear in the proof. This gives an order

of magnitude speedup in applications where cores or proofs have to be produced frequently [BKO+07, Bie08], and of course even more in the context of sophisticated (e.g., iterative) core/proof minimization techniques.

Since we also need proofs and cores from long runs, we needed to go beyond the current in-memory technology. Moreover, we use SAT solvers inside other systems (e.g., for SAT Modulo Theories) where memory for the SAT solver is more limited, All this will become even more important if (multicore) processor performance continues to grow faster than memory capacity. Here we describe a new and better in-memory method, and give a careful experimental comparison with the previous ones, which is non-trivial, since, for assessing different data structures and algorithms for SAT, it is crucial to develop implementations of each one of them, based on the same SAT solver, and *in such a way that the search performed by the SAT solver is always identical*. All software sources and benchmarks used here can be found at `www.lsi.upc.edu/~rasin`.

Our new in-memory algorithm, described in Section 4.2, keeps only the potentially needed parent information. The idea is to keep for each clause also a counter of how many of its children do have some active descendant. If it becomes zero the parent information can be removed (we have recently seen that the use of *reference counters* is suggested in [Bie08], but we do not know how similar this may be and no implementation exists). Here we show that (i) when implemented carefully, the overhead on the SAT solver time is still essentially negligible (around 5 per cent, similar to Biere's approach) and (ii) the memory usage frequently grows significantly slower. As the figure below shows, and as expected, in Biere's approach memory usage always grows linearly in the number of conflicts (or more, since parents lists get longer in longer runs), and hence also in his optimized Delta Encoding, which compresses parents lists up to four times [Bie08]. In our ChildCount approach, performing exactly the same search on this instance (`goldb-heqc-rotmul`; cf. Section 4.3 for many more experimental results), one can see in the figure that on this particular example memory usage grows much slower and even tends to stabilize. Skews in the plot correspond to clause deletion phases of the solver.



2 Short Overview on DPLL Algorithms for SAT

For self-containedness of the paper, here we give a short overview on DPLL based on the abstract presentation of [NOT06]. Let P be a fixed finite set of propositional symbols. If $p \in P$, then p is an *atom* and p and $\neg p$ are *literals* of P. The *negation* of a literal l, written $\neg l$, denotes $\neg p$ if l is p, and p if l is $\neg p$. A *clause* is a disjunction of literals $l_1 \vee \ldots \vee l_n$. A *unit clause* is a clause consisting of a single literal. A (CNF) *formula* is a conjunction of one or more clauses $C_1 \wedge \ldots \wedge C_n$. A (partial truth) *assignment* M is a set of literals such that $\{p, \neg p\} \subseteq M$ for no p. A literal l is *true* in M if $l \in M$, is *false* in M if $\neg l \in M$, and is *undefined* in M otherwise. A literal is *defined* in M if it is either true or false in M. A clause C is true in M if at least one of its literals is true in M. It is false in M if all its literals are false in M, and it is undefined in M otherwise. A formula F is true in M, or *satisfied* by M, denoted $M \models F$, if all its clauses are true in M. In that case, M is a *model* of F. If F has no models then it is *unsatisfiable*. If F and F' are formulas, we write $F \models F'$ if F' is true in all models of F. Then we say that F' *is entailed by* F, or is a *logical consequence* of F. If C is a clause $l_1 \vee \ldots \vee l_n$, we write $\neg C$ to denote the formula $\neg l_1 \wedge \ldots \wedge \neg l_n$. A *state* of the DPLL procedure is a pair of the form $M \parallel F$, where F is a finite set of clauses, and M is, essentially, a (partial) assignment. A literal l may be annotated as a *decision literal* (see below), writing it as l^{d}. A clause C is *conflicting* in a state $M \parallel F, C$ if $M \models \neg C$. A DPLL procedure can be modeled by a set of rules over such states:

UnitPropagate :

$$M \parallel F, C \vee l \implies M\ l \parallel F, C \vee l \quad \textbf{if} \quad \begin{cases} M \models \neg C \\ l \text{ is undefined in } M \end{cases}$$

Decide :

$$M \parallel F \implies M\ l^{\mathsf{d}} \parallel F \quad \textbf{if} \quad \begin{cases} l \text{ or } \neg l \text{ occurs in a clause of } F \\ l \text{ is undefined in } M \end{cases}$$

Fail :

$$M \parallel F, C \implies \mathit{Fail} \quad \textbf{if} \quad \begin{cases} M \models \neg C \\ M \text{ contains no decision literals} \end{cases}$$

Backjump :

$$M\ l^{\mathsf{d}}\ N \parallel F, C \implies M\ l' \parallel F, C \quad \textbf{if} \quad \begin{cases} M\ l^{\mathsf{d}}\ N \models \neg C, \text{ and there is} \\ \text{some clause } C' \vee l' \text{ such that:} \\ \quad F, C \models C' \vee l' \text{ and } M \models \neg C', \\ \quad l' \text{ is undefined in } M, \text{ and} \\ \quad l' \text{ or } \neg l' \text{ occurs in } F \text{ or in } M\ l^{\mathsf{d}}\ N \end{cases}$$

Learn :

$$M \parallel F \implies M \parallel F, C \quad \textbf{if} \quad \begin{cases} \text{each atom of } C \text{ occurs in } F \text{ or in } M \\ F \models C \end{cases}$$

Forget :

$$M \parallel F, C \implies M \parallel F \quad \textbf{if} \quad \{ F \models C$$

For deciding the satisfiability of an input formula F, one can generate an arbitrary derivation $\emptyset \parallel F \implies \ldots \implies S_n$, where S_n is a final state (no rule

applies). Under simple conditions, this always terminates. Moreover, for every derivation like the above ending in a final state S_n, (i) F is unsatisfiable if, and only if, S_n is *Fail*, and (ii) if S_n is of the form $M \parallel F$ then M is a model of F (see [NOT06] for all details).

The UnitPropagate, Decide and Fail rules speak for themselves. The Backjump rule corresponds to what is done in Example 1 (here $C' \vee l'$ is the backjump clause) and the Learn rule corresponds to the addition of lemmas (clauses that are logical consequences), such as the backjump clause. Since a lemma is aimed at preventing future similar conflicts, when these conflicts are not very likely to be found again the lemma can be removed by the Forget rule. In practice, a lemma is removed when its *relevance* (see, e.g., [BS97]) or its *activity* level drops below a certain threshold; the activity can be, e.g., the number of times it becomes a unit or a conflicting clause [GN02].

3 Basic Algorithms, Only for Core Extraction

In this section we introduce and compare two basic algorithms that can be used for extracting unsatisfiable cores, but not unsatisfiability proofs.

3.1 First Algorithm: Marker Literals

As said, in this approach one adds to each initial clause C_i one distinct new *initial ancestor (IA)* marker literal, say, a positive literal y_i. These literals are set to false from the beginning, and hence the logical meaning of the clause set does not change.

Then the solver is run, but without applying to the y_i-literals the usual simplification technique of removing from all clauses the literals that are false at decision level zero (henceforth: *false literal deletion*). In every lemma that is generated, its subset of y_i-literals shows exactly the subset of the initial clauses it has been derived from. In such a run, unsatisfiability is then witnessed by the appearance of an "empty clause" built solely from y_i-literals, i.e., a clause of the form $y_{j_1} \vee \ldots \vee y_{j_k}$, indicating that $\{C_{j_1}, \ldots, C_{j_k}\}$ is an unsatisfiable core. Note that this technique can only be used for finding unsatisfiable cores, and not for generating a resolution proof, since the proof structure is lost.

The interesting aspect of this method is that it requires very little implementation effort. However, it leads to important inefficiencies in the SAT solver. Clauses can become extremely long, using large amounts of memory, and for clauses that without the y_i-literals would have been units or two-literal clauses this is no longer the case. This leads to an important loss of efficiency in, for instance, the unit propagation data structures and algorithms.

3.2 Second Algorithm: Initial Ancestor Lists

An obvious way for overcoming the shortcomings of the previous algorithm is by storing initial ancestor information at the meta level along with the clauses,

instead of adding dummy literals for this. Therefore in this second algorithm each clause has an attached list with the ID's of its initial ancestors. This reduces part of the overhead of the first algorithm. For example, unit clauses are really treated as such, and are not hidden due to the additional IA literals.

In most DPLL-based SAT solvers, unit clauses and two-literal clauses are not explicitly stored as such. Units are usually simply set to true in the assignment at decision level zero, whereas binary clauses are typically kept in an adjacency list data structure, i.e., for each literal l there is a list of literals $l_1 \dots l_n$, such that each $l \vee l_i$ is a binary clause. This is much faster and memory-efficient for unit propagation than the standard two-watched literal data structures that are used for longer clauses.

In the algorithm for core extraction given here, we also need to store the IA information for unit clauses and two-literal clauses. This is done here in a memory bank apart from the one of the other clauses. Since one- and two-literal clauses are never removed in our solver, neither is their IA information.

3.3 Experiments: The First Two Algorithms vs. Our Basic Solver

In the first table below we compare a basic version of our own Barcelogic SAT solver without proof or core extraction (column **Basic**) with the two algorithms described in this section (**marker lits** and **IA's**). Each one of these two algorithms is implemented on top of the basic version with the minimal amount of changes. In particular, binary clauses are still represented in their efficient special form and no unit propagation using longer clauses is done if there is any pending two-literal clause propagation.

As said, for the algorithm based on marker literals we had to turn off false literal deletion. For the IA algorithm, each time a clause $C \vee l$ with IA list L_1 gets simplified due the decision level zero literal $\neg l$ with IA list L_2, the new clause C gets the IA list $L_1 \cup L_2$. It turns out that the IA lists became long and memory consuming. Therefore for this first experiment also in the **IA's** algorithm we switched off false literal deletion, which slowed down the solver and also made it search differently with respect to the basic version, but it prevented memory outs. Also to prevent memory outs, we were doing very frequent clause deletion rounds: every 5000 conflicts we were deleting all zero-activity clauses. To make the comparison fairer, we also did this in the basic algorithm, for which this is not precisely its optimal setting.

Note that therefore all three versions of the solver perform a different search[1] and hence, due to "luck" a core-generating version could still be faster than the basic one on some particular benchmark. All experiments were run on a 2.66GHz Xeon X3230, giving each process a 1.8GB memory limit and a timeout limit of 90 minutes. Times are indicated in seconds, and time outs are marked here with TO. The table is split into two parts. The first part has the unsatisfiable problems from the qualification instance sets of the 2006 SAT Race (SAT-Race_TS_1

[1] Below there is a version of the IA algorithm *with* false literal detection that does perform the same search as the basic version.

and 2, see `fmv.jku.at/sat-race-2006`) taking between 5 and 90 minutes in our basic solver. The second part has much easier ones. In all experiments the unsatisfiability of the extracted cores has been verified with independent SAT solvers.

From the results of the table it follows that these techniques are not practical except for very simple problems.

Runtimes (seconds)			
Instance	**Basic**	**marker lits**	**IA's**
manol-pipe-cha05-113	448	5035	786
manol-pipe-f7idw	546	2410	1181
6pipe	717	TO	1324
manol-pipe-g10idw	830	4171	2299
manol-pipe-c7idw	1534	TO	3701
manol-pipe-c10b	1938	TO	3926
manol-pipe-g10b	1969	TO	5365
manol-pipe-c6bid_i	2219	TO	4253
manol-pipe-g10ni	2419	TO	4412
manol-pipe-g10nid	2707	TO	TO
manol-pipe-c6nidw_i	2782	TO	TO
velev-dlx-uns-1.0-05	3306	1028	TO
goldb-heqc-frg2mul	3891	TO	TO
7pipe_q0_k	4184	TO	TO
manol-pipe-g10bidw	4650	TO	TO
goldb-heqc-i8mul	4911	TO	TO
hoons-vbmc-s04-06	TO	4543	TO
2dlx-cc-mc-ex-bp-f	1.81	2.91	1.35
3pipe-1-ooo	1.45	1.91	0.71
3pipe-3-ooo	1.92	3.53	1.59
4pipe-1-ooo	3.56	8.77	4.57
4pipe-3-ooo	5.38	11.67	5.36
4pipe-4-ooo	6.90	20.35	7.31
4pipe	8.15	33.64	14.82
5pipe-1-ooo	11.32	20.52	12.51
5pipe-2-ooo	10.31	18.98	14.33
5pipe-4-ooo	21.41	52.64	54.54
cache.inv14.ucl.sat.chaff.4.1.bryant	13.36	75.23	18.85
ooo.tag14.ucl.sat.chaff.4.1.bryant	7.05	6.78	7.96
s1841184384-of-bench-sat04-984.used-as.sat04-992	2.07	4.62	1.97
s57793011-of-bench-sat04-724.used-as.sat04-737	9.10	66.05	10.36
s376420895-of-bench-sat04-984.used-as.sat04-1000	2.50	5.48	2.28

It is well-known that DPLL-based SAT solvers are extremely sensitive in the sense that any small change (e.g., in the heuristic or in the order in which input clauses or their literals are given) causes the solver to search differently, which in

turn can cause dramatic changes in the runtime on a given instance. Therefore, most changes in SAT solvers are hard to assess, as they can only be evaluated by running a statistically significant amount of problems and measuring aspects like runtime averages. For this reason, all experiments mentioned from now on in this paper have been designed in such a way that for each method for proof/core extraction our solver performs *exactly the same search* (which was impossible in the algorithm with marker literals). This allows us to measure precisely the overhead in runtime and memory consumption due to proof/core generation bookkeeping.

The following table compares our basic solver on the easy problems with the IA's method, in runtime and in memory consumption. Here MO denotes memory out (> 1.8 GB). The difference in times with the previous table comes from the fact that here the setting of the solver is the standard one, with less frequent clause deletion phases, and with false literal deletion. As said, false literal deletion makes the IA's method even more memory consuming and also slower, as longer lists of parents have to be merged.

As we can see, usually only on the very simple problems the runtimes are comparable. As soon as more than few seconds are spent in the basic version, not only does the memory consumption explode, but also the runtime due to the bookkeeping (essentially, computing the union of long parents lists and copying them).

Basic vs IA's (same search, **T**ime in seconds, **M**emory in MB)				
Instance	**T Basic**	**M Basic**	**T IA's**	**M IA's**
2dlx-cc-mc-ex-bp-f	1.64	3	4.17	298
3pipe-1-ooo	1.35	3	2.07	122
3pipe-3-ooo	1.78	5	3.99	215
4pipe-1-ooo	3.98	14	22.76	843
4pipe-3-ooo	4.88	13	30.08	1175
4pipe-4-ooo	7.14	19	36.14	MO
4pipe	11.35	47	32.80	1106
5pipe-1-ooo	10.52	24	55.53	MO
5pipe-2-ooo	10.30	23	50.92	MO
5pipe-4-ooo	33.08	65	42.87	MO
cache.inv14.ucl.sat.chaff...	12.75	5	39.43	MO
ooo.tag14.ucl.sat.chaff.4...	6.21	3	9.12	612
s1841184384-of-bench-sat0...	1.83	1	1.86	51
s57793011-of-bench-sat04-...	7.75	32	8.47	74
s376420895-of-bench-sat04...	1.99	1	2.37	89

4 Algorithms for Extracting Proofs and Cores

Here we analyze more advanced algorithms that are not only able to extract unsatisfiable cores, but also resolution proof traces, i.e., the part of the trace that corresponds to the resolution proof.

4.1 In-Memory Parent Information

We now consider the in-memory method, a simpler version of which is implemented in the PicoSAT solver [Bie08]. Here, along with each clause the following additional information is stored: its ID, its list of immediate parents' ID's, and what we call its *is-parent bit*, saying whether this clause has generated any children itself or not. The parents list is what one would write to the trace in the [ZM03] technique. Each time a new lemma is generated, it gets a new ID, its is-parent bit is initialized to false, the ID's of its parents are collected and attached to it, and the is-parent bit of each one of its parents is set to true. In this approach, the parent information of a deleted clause (by application of the Forget rule during the clause deletion phase of the SAT solver) is removed only if its is-parent bit is false.

Once the empty clause is generated (i.e., a conflict at level zero appears), one can recover the proof by working backwards from it (without the need of traversing the whole trace, and on disk, as in [ZM03]).

In our implementation of this method, *unlike what is done in PicoSAT*, we maintain the special-purpose two-literal clause adjacency-list representation also when the solver is in proof-generation mode. Hence the performance slowdown with respect to our basic reference solver corresponds *exactly* to the overhead due to the bookkeeping for proof generation. Our implementation treats all conflicts in a uniform way, including the one obtaining the empty clause. This in in contrast to what is done with the final decision-level-zero conflict in Zhang and Malik's trace format, which gets a non-uniform treatment in [ZM03] (in fact, the explanations given in the introduction correspond to our simplified uniform view where the empty clause has its conflict analysis like any other clause).

The parents lists of units and binary clauses are stored in a separate memory zone, as we also did for the IA's method. Unit and binary clauses are never deleted in our solver. Essentially, at SAT solving time (more precisely, during conflict analysis) what is required is a direct access to the ID of a given clause. For unit and binary clauses we do this by hashing (for the larger clauses this is not necessary, since the clause, along with all its information and literals, is already being accessed during conflict analysis). At proof extraction time, one needs direct access to the parent list corresponding to a given clause ID. This we do by another hash table that only exists during proof extraction.

4.2 Our New Method with Child Count

The idea we develop in this section is the following: instead of just an is-parent bit, we keep along with each clause a counter, called the *childcounter*, of how many of its children have some *active* descendant. Here a clause is considered active if it participates in the DPLL derivation rules that are implemented in the SAT solver. In our solver, that is the case if it has less than three literals (these clauses are never deleted in our solver) or if it has at least three literals and has not been removed by the Forget rule (i.e., it is being *watched* in the two-watched literal data structure for unit propagation [MMZ+01]).

If the childcounter becomes zero also the parent information can be removed, since this clause can never appear in a proof trace of the empty clause (obtained from active clauses only). Note that this is a recursive process: each time a clause C is selected for deletion, i.e., when C goes from active to non-active, if its childcounter is zero then a recursive `childcounter-update`(C) process starts:

For each parent clause PC of C,

1. Decrease by one the childcounter of PC.
2. If the childcounter now becomes zero and PC is non-active, then do `childcounter-update`(PC).
3. Delete all information of C.

We have again implemented this method on top of our basic Barcelogic solver, and again we have done this in such a way that the search is not affected, i.e., again the additional runtime and memory consumption with respect to our basic solver correspond exactly to the overhead due to the bookkeeping for proof generation.

As before, during conflict analysis again we need to add the parent's ID's to the parent list of the new lemma, but now, in addition, the childcounters of these parents are increased. For this, as before, we use hashing to retrieve the ID of parent clauses with less than three literals. For the parent clauses with at least three literals this is not necessary, since these clauses, along with all their information and literals, are already being accessed during conflict analysis.

The main additional implementation issue is that now during the clause deletion phase, when doing `childcounter-update`(C), given the ID of an (active or non-active) clause, we may need access its information (their childcounters and parent lists). For this we use an additional hash table, which supposes only a negligible time overhead. Note that the clause deletion phase is not invoked very frequently and takes only a small fraction of the runtime of the SAT solver.

4.3 Experiments

We have run experiments with the same unsatisfiable instances as before (the harder ones): from the qualification instance sets of the 2006 SAT Race (SAT-Race_TS_1 and 2), the ones taking between 250 seconds and 90 minutes. Here again we run our solver in its standard settings, with false literal deletion and less frequent clause deletion phases.

In all experiments the correctness of the extracted proofs has been verified with the TraceCheck tool, see [Bie08] and `fmv.jku.at/tracecheck`, and a simple log information has been used to verify that indeed exactly the same SAT solving search was taking place in all versions.

Time consumption is analyzed in the next table (where instances are ordered by runtime) which has the following columns: **basic**: our basic SAT solver without proof/core generation, **Biere**: the same solver extended with Biere's in-memory core generation, **Biere-b**: the same, also extended with is-parent bit, **disk**: the basic solver writing traces to disk, as in [ZM03], **Child**: our method

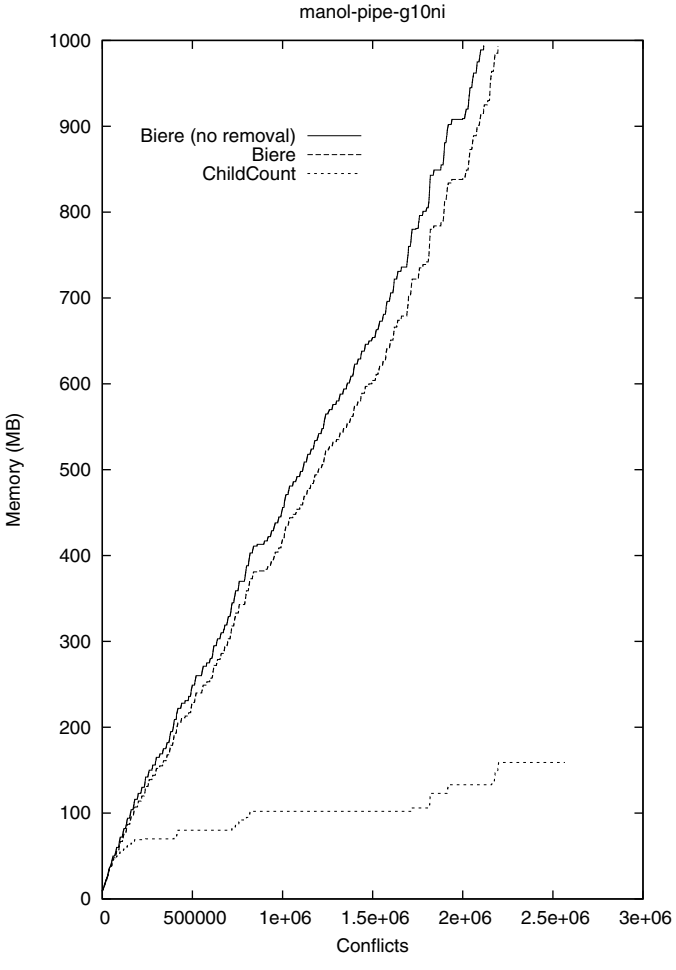
with child count. Columns "solve" include just the solving time (all version performing exactly the same search), and "slv+tr" includes as well the time needed for traversing the in-memory data structures and writing to disk the part of the trace that contains the unsatisfiability proof.

The entries labelled "MO" correspond to "Memory Out", which means more than 1.8GB. The entries labelled "FO" for the "disk" column correspond to more than 2GB, which produced a "file too large" error (to be eliminated in the final version of this paper).

Time (s)								
Instance	**basic**	**Biere**		**Biere-b**		**disk**	**Child**	
	solve	solve	slv+tr	solve	slv+tr	solve	solve	slv+tr
manol-pi-cha05-113	254	265	269	265	269	273	267	271
manol-pipe-f7idw	257	268	270	268	269	279	272	273
manol-pipe-c7idw	348	362	364	361	363	372	365	367
manol-pipe-g10idw	412	433	444	432	443	453	438	444
manol-pipe-c10b	527	550	561	546	558	567	555	564
goldb-heqc-i8mul	577	601	644	604	648	635	611	648
velev-dlx-uns-1.0-5	696	729	735	731	736	738	727	729
manol-pipe-c6bid_i	748	780	790	771	780	800	788	794
6pipe	785	846	858	844	856	850	854	861
velev-pipe-uns-1.1-7	829	928	948	930	949	941	931	940
manol-pipe-c6nidw_i	885	923	937	923	936	949	920	928
manol-pipe-g10nid	1030	1073	1080	1073	1079	1116	1071	1074
hoons-vbmc-s04-06	1053	1084	1099	1088	1103	1110	1107	1118
7pipe_q0_k	1551	1725	1776	1718	1764	1781	1751	1768
manol-pipe-g10bidw	1709	MO	MO	1774	1783	1856	1773	1777
manol-pipe-g10ni	1788	MO	MO	MO	MO	FO	2029	2033
manol-pipe-c7nidw	4059	MO	MO	MO	MO	FO	4209	4238
manol-pipe-c7bidw_i	4255	MO	MO	MO	MO	FO	4414	4445

The differences in runtime between our basic SAT solver without proof/core generation and its versions that do the necessary bookkeeping for in-memory proof/core generation are always very small, usually around five percent or less, and always less than the trace generation technique of [ZM03]. We conjecture that this is mainly because of the inefficiencies in writing to disk of the latter method (see below examples of the size of the traces that are written) since it requires less additional bookkeeping than the in-memory techniques. Note that our Childcount method in principle needs to do more work for generating the trace.

Much more important and interesting are the differences in memory usage. The plot we give below compares memory usage of three methods: (i) Biere's method without the is-parent bit (called "no removal" in the plot) i.e., where parent information is never deleted, (ii) Biere's method with the is-parent bit as explained here in Section 4.1, and (iii) our method with Childcount. We do this for one of the instances that generate many conflicts.



As we can see in the table below (where "**Time**" refers to the runtime of our basic SAT solver, and column "**Biere-b**" is the one with is-parent bit), the benefits of our Childcount methods are less important on examples that are solved generating fewer conflicts. The is-parent bit of Biere's methods has only a very limited impact. In the last two columns we also show the size of the whole DPLL trace on disk (**"full"**) produced by the method of [ZM03], and the size of its subset corresponding to the just the *proof trace* (**"proof"**), i.e., the proof of the empty clause, as it is generated by the methods Biere, Biere-b, and Childcount (which all three produce exactly the same proof trace in our implementations). Since the entire DPLL trace is usually much larger than just the proof trace, the in-memory methods are also faster if one writes to disk the proof trace once the unsatisfiability has been detected (although for many applications, such as core minimization, this is not needed).

In these implementations we have not considered compression methods such as Biere's Delta Encoding, which compresses parents lists up to four times [Bie08], since this is a somewhat orthogonal issue that can be applied (or not) to both methods.

Instance	Num. cnflcts	Time (s)	Memory Usage (MB) Biere	Biere-b	Child	Trace (MB) full	proof
velev-dlx-uns-1.0-05	199390	696	239	234	229	226	30
manol-pipe-f7idw	333275	257	140	127	78	183	24
manol-pipe-cha5-113	336968	254	228	218	167	218	112
goldb-heqc-i8mul	397702	577	MO	972	947	1002	937
manol-pipe-g10idw	423079	412	434	412	285	550	191
manol-pipe-c10b	530022	527	347	330	240	452	258
manol-pipe-c7idw	536341	348	209	192	141	217	42
manol-pipe-c6bid-i	1123035	748	393	347	187	543	166
manol-pipe-c6nidw-i	1256752	885	488	436	252	671	226
hoons-vbmc-s04-06	1301190	1053	320	309	228	358	322
manol-pipe-g10nid	1327600	1030	613	557	144	986	82
6pipe	1377876	785	519	502	418	433	205
velev-pipe-uns-1.1-7	1761066	829	447	409	210	751	260
manol-pipe-g10bidw	2250890	1709	MO	892	146	1679	100
manol-pipe-g10ni	2566801	1788	MO	MO	159	FO	113
7pipe-q0-k	3146242	1551	810	753	342	1381	472
manol-pipe-c7nidw	3585110	4059	MO	MO	613	FO	692
manol-pipe-c7bidw-i	4011227	4255	MO	MO	643	FO	761

5 Conclusions and Future Work

We have carried out a systematic and careful implementation of different methods for in-memory unsatisfiable core and proof generation. Regarding the two simpler methods for generating cores, our IA technique is indeed slightly more efficient than the one based on marker literals, but none of both is useful for instances on which our solver (using its default settings) takes more than few seconds. We have also shown that the techniques for generating cores and proofs explained in Section 4 are applicable to large SAT solving runs, and moreover allow one to keep the standard setting of the solver without a significant overhead in runtime.

Our experiments clearly show that our Childcount technique makes it possible to go significantly beyond previous in-memory techniques in terms of memory requirements. We plan to implement it in combination with Biere's Delta Encoding compression technique, which will make it possible to handle even longer DPLL runs or use even less memory. We also plan to use the basic underlying algorithms given here inside algorithms for core-minimization and for applications using cores (which are both outside the scope of this paper).

References

[Bie08] Biere, A.: PicoSAT essentials, Private communication. Journal on Satisfiability, Boolean Modeling and Computation (submitted, 2008)

[BKO+07] Bryant, R.E., Kroening, D., Ouaknine, J., Seshia, S.A., Strichman, O., Brady, B.A.: Deciding bit-vector arithmetic with abstraction. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 358–372. Springer, Heidelberg (2007)

[BS97] Bayardo Jr., R.J., Schrag, R.C.: Using CSP look-back techniques to solve real-world SAT instances. In: Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI 1997), Providence, Rhode Island, pp. 203–208 (1997)

[DLL62] Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Comm. of the ACM 5(7), 394–397 (1962)

[DP60] Davis, M., Putnam, H.: A computing procedure for quantification theory. Journal of the ACM 7, 201–215 (1960)

[FM06] Fu, Z., Malik, S.: On solving the partial max-sat problem. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 252–265. Springer, Heidelberg (2006)

[GN02] Goldberg, E., Novikov, Y.: BerkMin: A fast and robust SAT-solver. In: Design, Automation, and Test in Europe (DATE 2002), pp. 142–149 (2002)

[Hak85] Haken, A.: The intractability of resolution. Theor. Comput. Sci. 39, 297–308 (1985)

[Jac02] Jackson, D.: Alloy: a lightweight object modelling notation. ACM Trans. Softw. Eng. Methodol. 11(2), 256–290 (2002)

[MMZ+01] Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: Proc. 38th Design Automation Conference (DAC 2001) (2001)

[MSP08] Marques-Silva, J., Planes, J.: Algorithms for maximum satisfiability using unsatisfiable cores. In: Proceedings of Design, Automation and Test in Europe (DATE 2008), pp. 408–413 (2008)

[MSS99] Marques-Silva, J., Sakallah, K.A.: GRASP: A search algorithm for propositional satisfiability. IEEE Trans. Comput. 48(5), 506–521 (1999)

[NOT06] Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT Modulo Theories: from an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). Journal of the ACM 53(6), 937–977 (2006)

[ZM03] Zhang, L., Malik, S.: Validating SAT Solvers Using an Independent Resolution-Based Checker: Practical Implementations and Other Applications. In: 2003 Design, Automation and Test in Europe Conference (DATE 2003), pp. 10880–10885. IEEE Computer Society, Los Alamitos (2003)

Justification-Based Local Search with Adaptive Noise Strategies

Matti Järvisalo, Tommi Junttila, and Ilkka Niemelä

Helsinki University of Technology (TKK)
Department of Information and Computer Science, P.O. Box 5400, FI-02015 TKK, Finland
{matti.jarvisalo,tommi.junttila,ilkka.niemela}@tkk.fi

Abstract. We study a framework called BC SLS for a novel type of stochastic local search (SLS) for propositional satisfiability (SAT). Aimed specifically at solving real-world SAT instances, the approach works directly on a non-clausal structural representation for SAT. This allows for don't care detection and justification guided search heuristics in SLS by applying the circuit-level SAT technique of justification frontiers. In this paper we extend the BC SLS approach first by developing generalizations of BC SLS which are probabilistically approximately complete (PAC). Second, we develop and study adaptive noise mechanisms for BC SLS, including mechanisms based on dynamically adapting the waiting period for noise increases. Experiments show that a preliminary implementation of the novel adaptive, PAC generalization of the method outperforms a well-known CNF level SLS method with adaptive noise (AdaptNovelty+) on a collection of structured real-world SAT instances.

1 Introduction

While stochastic local search techniques (SLS) such as [1,2,3,4,5] are very efficient in solving hard randomly generated SAT instances, a major challenge is to improve SLS on structural problems by efficiently handling variable dependencies [6]. In this paper we extend a recent non-clausal stochastic local search (SLS) method, BC SLS [7], which applies similar ideas as typical in clausal SLS methods but differs in many crucial aspects. In particular, BC SLS combines techniques from structure-based complete DPLL style non-clausal algorithms [8,9,10,11]. Aimed specifically at solving real-world SAT instances, BC SLS works directly on a non-clausal structural representation for SAT. This allows for adopting don't cares [12] and justification guided search heuristics in SLS by applying ideas from the circuit-level SAT technique of justification frontiers [10]. For a discussion of the relationship between the basic BC SLS method and both CNF level and other non-clausal SLS methods, such as [13,14,15], see [7].

In this work we adopt the basic ingredients of local search—the notions of a configuration and a move, the objective function, and the stopping criterion—from BC SLS, and extend the approach. In more detail, we develop generalizations of BC SLS which (i) are *probabilistically approximately complete* (PAC) [16], and which (ii) *exploit adaptive noise mechanisms* within the framework.

It has been observed that the performance of CNF level SLS methods, such as those in the WalkSAT family, varies greatly depending on the chosen fixed noise parameter

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 31–46, 2008.

setting [3,4]. We show that the same phenomenon is present also in BC SLS. In the case of CNF level SLS, in order to avoid manual noise tuning this has led to the development of automatic noise level mechanisms based on probing techniques for selecting a fixed noise parameter setting before actual search [17], or by adaptively tuning the noise level during search [4]. Here we adapt latter techniques to the BC SLS framework. However, we discover that compared to the parameter values for adapting noise used in CNF level SLS methods, radically different settings are required in BC SLS. We then show how to adjust this technique for BC SLS for better performance. In addition to the adaptive noise mechanism based on a static waiting period for noise increments, we suggest an alternative based on dynamic waiting periods that depend more on the current state of the search. While maintaining similar performance, the application of dynamic waiting periods gives the possibility of dismissing the fixed constant used in the typical adaptive noise mechanism based on a static waiting periods.

Applying a novel adaptive noise strategy for BC SLS, we show experimentally that a preliminary implementation of an adaptive PAC variant of the BC SLS method outperforms a fine-tuned implementation of the CNF level SLS method AdaptNovelty+ on a collection of structured real-world SAT instances.

This paper is organized as follows. First we define Boolean circuits and central concepts related to justifications and don't cares (Sect. 2). The justification-based non-clausal SLS framework is described in Sect. 3, with analysis of probabilistically approximately completeness of different variants of the method (Sect. 3.1). Section 4 is focused on developing adaptive noise mechanisms for the framework.

2 Constrained Boolean Circuits

Boolean circuits offer a natural non-clausal representation for *arbitrary* propositional formulas in a compact DAG-like structure with *subformula sharing*. Rather than translating circuits to CNF for solving the resulting SAT instance by local search, in this work we will work directly on the Boolean circuit representation.

A *Boolean circuit* over a finite set G of *gates* is a set $\mathcal{C}$ of equations of form $g := f(g_1, \ldots, g_n)$, where $g, g_1, \ldots, g_n \in G$ and $f : \{\mathbf{f}, \mathbf{t}\}^n \to \{\mathbf{f}, \mathbf{t}\}$ is a Boolean function, with the additional requirements that (i) each $g \in G$ appears at most once as the left hand side in the equations in $\mathcal{C}$, and (ii) the underlying directed graph

$$\langle G, E(\mathcal{C}) = \{\langle g', g\rangle \in G \times G \mid g := f(\ldots, g', \ldots) \in \mathcal{C}\}\rangle$$

is acyclic. If $\langle g', g\rangle \in E(\mathcal{C})$, then g' is a *child* of g and g is a *parent* of g'. The *descendant* and *ancestor* relations are defined in the usual way as the transitive closures of the child and parent relations, respectively. If $g := f(g_1, \ldots, g_n)$ is in $\mathcal{C}$, then g is an f-gate (or of type f), otherwise it is an *input gate*. The set of input gates in $\mathcal{C}$ is denoted by $\mathsf{inputs}(\mathcal{C})$. A gate with no parents is an *output gate*. An *assignment* for $\mathcal{C}$ is a (possibly partial) function $\tau : G \to \{\mathbf{f}, \mathbf{t}\}$. A total assignment τ for $\mathcal{C}$ is *consistent* if $\tau(g) = f(\tau(g_1), \ldots, \tau(g_n))$ for each $g := f(g_1, \ldots, g_n)$ in $\mathcal{C}$. A circuit $\mathcal{C}$ has $2^{|\mathsf{inputs}(\mathcal{C})|}$ consistent total assignments.

A *constrained Boolean circuit* $\mathcal{C}^\alpha$ is a pair $\langle \mathcal{C}, \alpha\rangle$, where $\mathcal{C}$ is a Boolean circuit and α is an assignment for $\mathcal{C}$. With respect to a constrained circuit $\mathcal{C}^\alpha$, each $\langle g, v\rangle \in \alpha$ is a

constraint, and g is *constrained* to v if $\langle g, v\rangle \in \alpha$. A total assignment τ for $\mathcal{C}$ *satisfies* $\mathcal{C}^\alpha$ if (i) τ is consistent with $\mathcal{C}$, and (ii) respects the constraints: $\tau \supseteq \alpha$. If some total assignment satisfies $\mathcal{C}^\alpha$, then $\mathcal{C}^\alpha$ is *satisfiable* and otherwise *unsatisfiable*. In this work we consider Boolean circuits in which the following Boolean functions are available as gate types.

- $\text{NOT}(v)$ is $\mathbf{t}$ iff v is $\mathbf{f}$.
- $\text{OR}(v_1, \ldots, v_n)$ is $\mathbf{t}$ iff at least one of $v_1, \ldots, v_n$ is $\mathbf{t}$.
- $\text{AND}(v_1, \ldots, v_n)$ is $\mathbf{t}$ iff all $v_1, \ldots, v_n$ are $\mathbf{t}$.
- $\text{XOR}(v_1, v_2)$ is $\mathbf{t}$ iff exactly one of v_1, v_2 is $\mathbf{t}$.

However, notice that the techniques developed in this paper can be adapted for a wider range of types. In order to keep the presentation and algorithms simpler, we assume that constraints only appear in the output gates of constrained circuits. Any circuit can be rewritten into such a normal form by using the rules in [8].

Figure 1 shows a Boolean circuit for a full-adder with the constraint that the carry-out bit c_1 is $\mathbf{t}$. Formally the circuit is defined as $\mathcal{C} = \{c_1{:=}\text{OR}(t_1, t_2), t_1{:=}\text{AND}(t_3, c_0), o_0{:=}\text{XOR}(t_3, c_0), t_2{:=}\text{AND}(a_0, b_0), t_3{:=}\text{XOR}(a_0, b_0)\}$, and $\alpha = \{\langle c_1, \mathbf{t}\rangle\}$. A satisfying total assignment for it is $\{\langle c_1, \mathbf{t}\rangle, \langle t_1, \mathbf{t}\rangle, \langle o_0, \mathbf{f}\rangle, \langle t_2, \mathbf{f}\rangle, \langle t_3, \mathbf{t}\rangle, \langle a_0, \mathbf{t}\rangle, \langle b_0, \mathbf{f}\rangle, \langle c_0, \mathbf{t}\rangle\}$. The *restriction* of an assignment τ to a set $G' \subseteq G$ of gates is defined as usual: $\tau|_{G'} = \{\langle g, v\rangle \in \tau \mid g \in G'\}$. Given a non-input gate $g := f(g_1, \ldots, g_n)$ and a value $v \in \{\mathbf{f}, \mathbf{t}\}$, a *justification* for the pair $\langle g, v\rangle$ is a partial assignment $\sigma : \{g_1, \ldots, g_n\} \to \{\mathbf{f}, \mathbf{t}\}$ to the children of g such that $f(\tau(g_1), \ldots, \tau(g_n)) = v$ holds for all extensions $\tau \supseteq \sigma$. That is, the values assigned by σ to the children of g are enough to force g to have the value v. A gate g is *justified in an assignment* τ if it is assigned, i.e. $\tau(g)$ is defined, and (i) it is an input gate, or (ii) $g := f(g_1, \ldots, g_n) \in \mathcal{C}$ and $\tau|_{\{g_1,\ldots,g_n\}}$ is a justification for $\langle g, \tau(g)\rangle$. For example, consider the gate t_1 in Fig. 1. The possible justifications for $\langle t_1, \mathbf{f}\rangle$ are $\{\langle t_3, \mathbf{f}\rangle\}$, $\{\langle t_3, \mathbf{f}\rangle, \langle c_0, \mathbf{t}\rangle\}$, $\{\langle t_3, \mathbf{f}\rangle, \langle c_0, \mathbf{f}\rangle\}$, $\{\langle c_0, \mathbf{f}\rangle\}$, and $\{\langle t_3, \mathbf{t}\rangle, \langle c_0, \mathbf{f}\rangle\}$; of these the first and fourth one are subset minimal ones. The gate t_1 is justified in the assignment $\tau = \{\langle c_1, \mathbf{t}\rangle, \langle t_1, \mathbf{f}\rangle, \langle o_0, \mathbf{t}\rangle, \langle t_2, \mathbf{t}\rangle, \langle t_3, \mathbf{f}\rangle, \langle a_0, \mathbf{t}\rangle, \langle b_0, \mathbf{t}\rangle, \langle c_0, \mathbf{t}\rangle\}$.

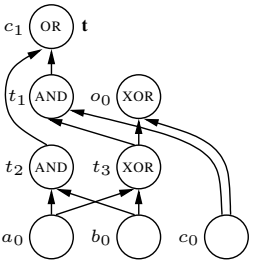


Fig. 1. A constrained circuit

A key concept in BC SLS is the *justification cone* $\mathsf{jcone}(\mathcal{C}^\alpha, \tau)$ for a constrained circuit $\mathcal{C}^\alpha$ under an assignment $\tau \supseteq \alpha$. The justification cone is defined recursively top-down in the circuit structure, starting from the constrained gates. Intuitively, the cone is the smallest set of gates which includes all constrained gates and, for each justified gate in the set, all the gates that participate in some subset minimal justification for the gate. More formally, $\mathsf{jcone}(\mathcal{C}^\alpha, \tau)$ is the smallest one of those sets S of gates which satisfy the following properties.

1. If $\langle g, v\rangle \in \alpha$, then $g \in S$.
2. If $g \in S$ and (i) g is a non-input gate, (ii) g is justified in τ, and (iii) $\langle g_i, v_i\rangle \in \sigma$ for some subset minimal justification σ for $\langle g, \tau(g)\rangle$, then $g_i \in S$.

Notice that by this definition $\mathsf{jcone}(\mathcal{C}^\alpha, \tau)$ is unambiguously defined.

As another key concept, the *justification frontier* of $\mathcal{C}^\alpha$ under τ, is the "bottom edge" of the justification cone, i.e. those gates in the cone that are not justified:

$$\mathsf{jfront}(\mathcal{C}^\alpha, \tau) = \{g \in \mathsf{jcone}(\mathcal{C}^\alpha, \tau) \mid g \text{ is not justified in } \tau\}.$$

A gate g is *interesting* in τ if it belongs to the frontier $\mathsf{jfront}(\mathcal{C}^\alpha, \tau)$ or is a descendant of a gate in it; the set of all gates that are interesting in τ is denoted by $\mathsf{interest}(\mathcal{C}^\alpha, \tau)$. A gate g is an (*observability*) *don't care* if it is neither interesting nor in the justification cone $\mathsf{jcone}(\mathcal{C}^\alpha, \tau)$. For instance, consider the constrained circuit $\mathcal{C}^\alpha$ in Fig. 1. Under the assignment $\tau = \{\langle c_1, \mathbf{t}\rangle, \langle t_1, \mathbf{t}\rangle, \langle o_0, \mathbf{f}\rangle, \langle t_2, \mathbf{f}\rangle, \langle t_3, \mathbf{t}\rangle, \langle a_0, \mathbf{f}\rangle, \langle b_0, \mathbf{f}\rangle, \langle c_0, \mathbf{t}\rangle\}$, the justification cone $\mathsf{jcone}(\mathcal{C}^\alpha, \tau)$ is $\{c_1, t_1, t_3, c_0\}$, the justification frontier $\mathsf{jfront}(\mathcal{C}^\alpha, \tau)$ is $\{t_3\}$, $\mathsf{interest}(\mathcal{C}^\alpha, \tau) = \{t_3, a_0, b_0\}$, and the gates t_2 and o_0 are don't cares.

As observed in [7] if the justification frontier $\mathsf{jfront}(\mathcal{C}^\alpha, \tau)$ is empty for some total assignment τ, then the constrained circuit $\mathcal{C}^\alpha$ is satisfiable. When $\mathsf{jfront}(\mathcal{C}^\alpha, \tau)$ is empty, a satisfying assignment can be obtained by (i) restricting τ to the input gates appearing in the justification cone, i.e. to the gate set $\mathsf{jcone}(\mathcal{C}^\alpha, \tau) \cap \mathsf{inputs}(\mathcal{C})$, (ii) assigning other input gates arbitrary values, and (iii) recursively evaluating the values of non-input gates. Thus, whenever $\mathsf{jfront}(\mathcal{C}^\alpha, \tau)$ is empty, we say that τ *de facto satisfies* $\mathcal{C}^\alpha$. For example, the assignment $\{\langle c_1, \mathbf{t}\rangle, \langle t_1, \mathbf{f}\rangle, \langle o_0, \mathbf{f}\rangle, \langle t_2, \mathbf{t}\rangle, \langle t_3, \mathbf{t}\rangle, \langle a_0, \mathbf{t}\rangle, \langle b_0, \mathbf{t}\rangle, \langle c_0, \mathbf{t}\rangle\}$ de facto satisfies the constrained circuit $\mathcal{C}^\alpha$ in Fig. 1; a satisfying assignment obtained by the procedure above is $\{\langle c_1, \mathbf{t}\rangle, \langle t_1, \mathbf{f}\rangle, \langle o_0, \mathbf{f}\rangle, \langle t_2, \mathbf{t}\rangle, \langle t_3, \mathbf{f}\rangle, \langle a_0, \mathbf{t}\rangle, \langle b_0, \mathbf{t}\rangle, \langle c_0, \mathbf{f}\rangle\}$. Also note that if a total truth assignment τ satisfies $\mathcal{C}^\alpha$, then $\mathsf{jfront}(\mathcal{C}^\alpha, \tau)$ is empty.

Translating Circuits to CNF. Each constrained Boolean circuit $\mathcal{C}^\alpha$ can be translated into an equi-satisfiable CNF formula $\mathsf{cnf}(\mathcal{C}^\alpha)$ by applying the standard "Tseitin translation". In order to obtain a small CNF formula, the idea is to introduce a variable $\tilde{g}$ for each gate g in the circuit, and then to describe the functionality of each gate with a set of clauses. For instance, an AND-gate $g := \text{AND}(g_1, \ldots, g_n)$ is translated into the clauses $(\neg\tilde{g} \vee \tilde{g}_1), \ldots, (\neg\tilde{g} \vee \tilde{g}_n)$, and $(\tilde{g} \vee \neg\tilde{g}_1 \vee \ldots \vee \neg\tilde{g}_n)$. The constraints are translated into unit clauses: introduce the clause $(\tilde{g})$ for $\langle g, \mathbf{t}\rangle \in \alpha$, and the clause $(\neg\tilde{g})$ for $\langle g, \mathbf{f}\rangle \in \alpha$.

A Note on Negations. As usual in many SAT algorithms, we will implicitly ignore NOT-gates of form $g := \text{NOT}(g_1)$; g and g_1 are always assumed to have the opposite values. Thus NOT-gates are, for instance, (i) "inlined" in the cnf translation by substituting $\neg\tilde{g}_1$ for $\tilde{g}$, and (ii) never counted in an interest set $\mathsf{interest}(\mathcal{C}^\alpha, \tau)$.

3 Justification-Based Non-clausal SLS

In the non-clausal method BC SLS [7] a configuration is described by a total truth assignment as in typical clausal SLS methods. However, the non-clausal method works directly on general propositional formulas represented as Boolean circuits, and hence a configuration is a total assignment on the gates of the Boolean circuit at hand. Moreover, the key elements of an SLS method – the notion of moves, the objective function, and the stopping criterion – are substantially different from the corresponding elements in clausal SLS methods.

In typical SLS methods for SAT the moves consist of individual flips on variable values in the current configuration. In BC SLS structural knowledge is exploited for making moves on gates: a typical move on a gate g flips the values of a subset of g's

children so that g becomes locally justified under the new truth assignment. Moreover, moves are focused on a particular subset of the gates, the justification frontier, which guides the search to concentrate on relevant parts of the instance exploiting *observability don't cares*. In typical clausal SLS methods the objective function measures the number of clauses that are falsified by the current truth assignment. In BC SLS the objective function is based on the concept of justification frontier and uses the set of interesting gates. The notion of a justification frontier leads to a early stopping criterion where the search can be halted when the circuit has been shown to be de facto satisfiable which often occurs before a total satisfying truth assignment has been found.

In this work we extend BC SLS in order to (i) achieve a *probabilistically approximately complete* (PAC) generalization of the method, and to (ii) *exploit adaptive noise mechanisms* within the framework. The resulting generalized framework is described as Algorithm 1. Given a constrained Boolean circuit $\mathcal{C}^\alpha$ the algorithm performs structural local search over the assignment space of *all* the gates in $\mathcal{C}$ (inner loop on lines 3–13). As typical, the *noise parameter* $p \in [0,1]$ controls the probability of making non-greedy moves (with $p = 0$ only greedy moves are made). Here we introduce an additional parameter $q \in [0,1]$ which leads to PAC variants of BC SLS. We will consider adaptive noise mechanisms for controlling the value of p during the search in Sect. 4.

Algorithm 1. Generalized BC SLS

Input: constrained Boolean circuit $\mathcal{C}^\alpha$, control parameters $p, q \in [0,1]$ for non-greedy moves
Output: a *de facto* satisfying assignment for $\mathcal{C}^\alpha$ or "*don't know*"
Explanations:
τ: current truth assignment on all gates with $\tau \supseteq \alpha$
δ: next move (a partial assignment)

1: **for** $try := 1$ to MAXTRIES$(\mathcal{C}^\alpha)$ **do**
2: $\quad\tau :=$ pick an assignment over all gates in $\mathcal{C}$ s.t. $\tau \supseteq \alpha$
3: $\quad$**for** $move := 1$ to MAXMOVES$(\mathcal{C}^\alpha)$ **do**
4: $\quad\quad$**if** $\mathsf{jfront}(\mathcal{C}^\alpha, \tau) = \emptyset$ **then return** τ
5: $\quad\quad$Select a random gate $g \in \mathsf{jfront}(\mathcal{C}^\alpha, \tau)$
6: $\quad\quad$**with** probability $(1-p)$ **do** *%greedy move*
7: $\quad\quad\quad\delta :=$ a random justification from those justifications for $\langle g, v\rangle \in \tau$ that minimize $\mathsf{cost}(\tau, \cdot)$
8: $\quad\quad$**otherwise** *%non-greedy move (with probability p)*
9: $\quad\quad\quad$**if** g is constrained in α **or with** probability q **do**
10: $\quad\quad\quad\quad\delta :=$ a random justification for $\langle g, v\rangle \in \tau$
11: $\quad\quad\quad$**else**
12: $\quad\quad\quad\quad\delta := \{\langle g, \neg\tau(g)\rangle\}$ *%flip the value of g*
13: $\quad\quad\tau := (\tau \setminus \{\langle g, \neg w\rangle \mid \langle g, w\rangle \in \delta\}) \cup \delta$
14: **return** "*don't know*"

For each of the MAXTRIES$(\mathcal{C}^\alpha)$ runs, MAXMOVES$(\mathcal{C}^\alpha)$ moves are made. As the *stopping criterion* we use the condition that the justification frontier $\mathsf{jfront}(\mathcal{C}^\alpha, \tau)$ is empty. As discussed in Section 2 if $\mathsf{jfront}(\mathcal{C}^\alpha, \tau)$ is empty, then $\mathcal{C}^\alpha$ is satisfiable and a satisfying truth assignment can be computed from τ. Notice that typically this stopping criterion is reached before all gates are justified in the current configuration τ.

Given the current configuration τ, we concentrate on making moves on gates in $\mathsf{jfront}(\mathcal{C}^\alpha, \tau)$ by randomly picking a gate g from this set. For a gate g and its current value v in τ, the possible *greedy moves* are induced by the justifications for $\langle g, v\rangle$. The idea is to minimize the *size of the interest set*. In other words, the value of the objective function for a move (justification) δ is $\mathsf{cost}(\tau, \delta) = |\mathsf{interest}(\mathcal{C}^\alpha, \tau')|$, where $\tau' = (\tau \setminus \{\langle g, \neg w\rangle \mid \langle g, w\rangle \in \delta\}) \cup \delta$. That is, the cost of a move δ is the size of the interest set in the configuration τ' where for the gates mentioned in δ we use the values in δ instead of those in τ. The move is then selected randomly from those justifications δ for $\langle g, v\rangle$ for which $\mathsf{cost}(\tau, \delta)$ is smallest over all justifications for $\langle g, v\rangle$.

During a *non-greedy move* (lines 9–12, executed with probability p), we introduce a new parameter q for guaranteeing the PAC property (for PAC proofs, see Section 3.1). For non-greedy moves, the control parameter q defines the probability of justifying the selected gate g by a randomly chosen justification from the set of all justifications for the value of g (this is a *non-greedy downward move*). With probability $(1 - q)$ the non-greedy move consists of inverting the value of *the gate g itself* (a *non-greedy upward move*). The idea in upward moves is to try to escape from possible local minima by more radically changing the justification front. In the special case when g is constrained, a random downward move is done with probability 1.

Notice that the size of the interest set gives an upper bound on the number of gates that still need to be justified (the descendants of the gates in the front). Following this intuition, by applying the objective function of minimizing the size of the interest set, the greedy moves drive the search towards the input gates. Alternatively, one could use the objective of minimizing the size of the justification frontier since moves are concentrated on gates in the frontier and since the search is stopped when the frontier is empty. However, we notice that the size of the interest set is more responsive to quantifying the changes in the configuration than the size of the justification frontier, as exemplified in Fig. 2. The size of the frontier typically drops rapidly close to zero percents from its starting value (the y axis is scaled to $[0, 1]$ in the figure), and after this remains quite stable until a solution is found. This is very similar to the typical behavior observed for objective functions based on the number of unsatisfied clauses in CNF level SLS methods [18]. In contrast, the size of the interest set can vary significantly without visible changes in the size of the justification frontier. Using the size of the interest set rather than the size of the justification frontier also resulted in better performance in preliminary experiments.

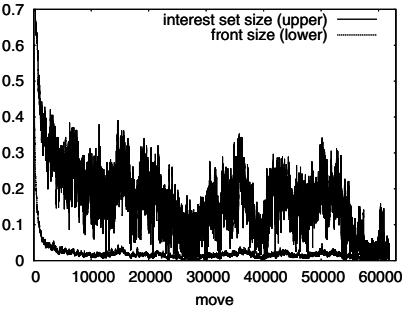


Fig. 2. Comparison of dynamics: sizes of interest set and justification frontier

3.1 On the PAC Property in BC SLS

We now analyze under which conditions BC SLS is PAC (*probabilistically approximately complete*) [16]. A CNF-level SLS SAT method S is PAC if, for any satisfiable

CNF SAT instance F and any initial configuration τ, the probability that S *eventually* finds a satisfying truth assignment for F starting from τ is 1 *without using restarts*, i.e., the number of allowed flips is set to infinity and the number of tries to one. A non-PAC SLS method is *essentially incomplete*. Examples of PAC CNF level SLS methods include GWSAT (with non-zero random walk probability) and UnitWalk, while GSAT, WalkSAT/TABU and Novelty (for arbitrary noise parameter setting) are essentially incomplete [16,19]. Here we adapt the definition of PAC to the context of BC SLS.

Definition 1. *BC SLS is PAC using fixed parameters p, q if, for any satisfiable constrained circuit $\mathcal{C}^\alpha$ and any initial configuration τ, the probability that BC SLS eventually finds a de facto satisfying assignment for $\mathcal{C}^\alpha$ starting from τ is 1 when setting* $\textsc{MaxTries}(\mathcal{C}^\alpha) = 1$ *and* $\textsc{MaxMoves}(\mathcal{C}^\alpha) = \infty$.

It turns out that for a PAC variant of BC SLS, both upward and downward non-greedy moves are needed.

Theorem 1. *The variant of BC SLS where non-greedy downward moves are allowed with probability q, where $0 < q < 1$, is PAC for any fixed noise parameter $p > 0$.*

Proof. Assume that $\mathcal{C}^\alpha$ is satisfiable, the current assignment is τ, and $\mathsf{jfront}(\mathcal{C}^\alpha, \tau) \neq \emptyset$. We show that by executing the inner loop (lines 3–13) at most $|G|$ times the algorithm reaches a de facto satisfying assignment with probability of at least

$$\left(\frac{1}{|G|} \cdot p \cdot \min(q \cdot \frac{1}{2^{|G|}}, 1-q)\right)^{|G|}.$$

First, take any satisfying assignment $\tau^\star$ for $\mathcal{C}^\alpha$. Recall that $\mathsf{jfront}(\mathcal{C}^\alpha, \tau^\star) = \emptyset$ by definition. Repeat the following until $\mathsf{jfront}(\mathcal{C}^\alpha, \tau) = \emptyset$.

1. If there is a gate g in the frontier $\mathsf{jfront}(\mathcal{C}^\alpha, \tau)$ such that $\tau(g) \neq \tau^\star(g)$, execute the line 12 that flips the value $\tau(g)$ to $\tau^\star(g)$. Note that g is not constrained by α as both $\tau, \tau^\star \supseteq \alpha$. Thus this step happens with the probability of at least $\frac{1}{|G|} \cdot p \cdot (1-q)$.
2. Otherwise the current assignment τ is such that all the gates in the justification cone and frontier under τ have the same values as in the satisfying truth assignment $\tau^\star$. Take a gate g in the frontier $\mathsf{jfront}(\mathcal{C}^\alpha, \tau)$. Now there is at least one child of g whose value differs in τ and $\tau^\star$. Execute the line 10 in a way that only flips the values of children of g whose values differ in τ and $\tau^\star$; the value of at least one such child is flipped. This step happens with the probability of at least $\frac{1}{|G|} \cdot p \cdot q \cdot \frac{1}{2^{|G|}}$, where the term $\frac{1}{2^{|G|}}$ comes from the fact that a gate always has less than $|G|$ children, and thus the probability of picking the desired justification is at least $\frac{1}{2^{|G|}}$.

As both steps above (i) flip the value of at least one gate to one in $\tau^\star$ and (ii) never flip a gate whose value already is the same as in $\tau^\star$, they are executed at most $|G|$ times: after this $\tau = \tau^\star$ and thus $\mathsf{jfront}(\mathcal{C}^\alpha, \tau) = \mathsf{jfront}(\mathcal{C}^\alpha, \tau^\star) = \emptyset$. Naturally, it may happen that $\mathsf{jfront}(\mathcal{C}^\alpha, \tau) = \emptyset$ earlier and the process terminates in fewer than $|G|$ steps; now τ is not necessary equal to $\tau^\star$ but is de facto satisfying anyway. Therefore, executing the lines 3–13 $|G|$ times transforms the current assignment into a de facto satisfying

assignment with probability of at least $\left(\frac{1}{|G|} \cdot p \cdot \min(q \cdot \frac{1}{2^{|G|}}, 1-q)\right)^{|G|}$. Since this is non-zero when $p > 0$ and $0 < q < 1$, BC SLS finds a satisfying assignment with probability one as $\text{MAXMOVES}(\mathcal{C}^\alpha)$ approaches infinity. □

Interestingly, at least for the gate types considered here, downward non-greedy moves can be restricted to *minimal* justifications without affecting Theorem 1.

However, if non-greedy moves are only allowed either (i) upwards or (ii) downwards, then BC SLS becomes essentially incomplete.

Theorem 2. *The variant of BC SLS where non-greedy moves are done only upwards (i.e. when $q = 0$) is essentially incomplete for any fixed noise parameter p.*

Proof. Consider the constrained circuit $\mathcal{C}^\alpha$ in Fig. 3; the subcircuit C_f is such that the gate d can evaluate both to **t** or **f**, depending on the values of the input gates, while C_g is a subcircuit that only allows the gate e to evaluate to **f**. Therefore the gate d must have the value **t** in any (de facto or standard) satisfying assignment. Furthermore, assume that the subcircuit C_g has fewer gates than C_f.

Assume that the current assignment τ assigns the gate d to **f** and that $\langle d, \mathbf{f}\rangle$ is not justified under τ. Now if $\tau(b) = \mathbf{f}$, the gate b cannot be in the frontier, and the inner loop (lines 3–13) of BC SLS cannot change the value of d to **t**. If $\tau(b) = \mathbf{t}$ (and thus b is in the justification cone), either (i) $\tau(e) = \mathbf{t}$ implying that d is a don't care and thus its value cannot be changed in the inner loop, or (ii) $\tau(e) = \mathbf{f}$ implying that b is in the frontier and the inner loop can pick an interest set size minimizing justification for b on line 7 (but random justification on line 10 is not in use as $q = 0$). In case (ii), as C_g has fewer gates than C_f and $\langle d, \mathbf{f}\rangle$ is not justified in τ, the greedy move will flip the value of e to **t** and leave d intact because the whole subcircuit C_f becomes a don't care and is removed from the interest set. To sum up, when $q = 0$ the inner loop cannot change the value of d and never finds a de facto satisfying assignment. □

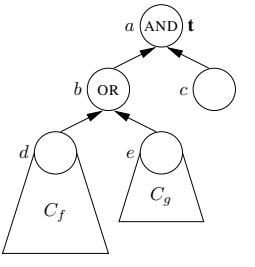


Fig. 3. A circuit

Theorem 3. *The variant of BC SLS where non-greedy moves are done only downwards (i.e. when $q = 1$) is essentially incomplete for any fixed noise parameter p.*

Proof. Consider again the constrained circuit $\mathcal{C}^\alpha$ in Fig. 3 with the assumption that the subcircuit C_f is such that the gate d can evaluate both to **t** or **f**, depending on the values of the input gates, while C_g is a subcircuit that only allows the gate e to evaluate to **f**. Suppose that the current assignment τ assigns b to **t**, d to **f**, and e to **t**. Now the gate b is not in the frontier. Because of this and the fact that the line 12 is never executed when $q = 1$, the (incorrect) value of e cannot be changed in the inner loop (lines 3–13) of BC SLS. Thus b never appears in the frontier and the (incorrect) value of the gate d cannot be changed during the execution of the inner loop. Thus a de facto satisfying assignment is never found. □

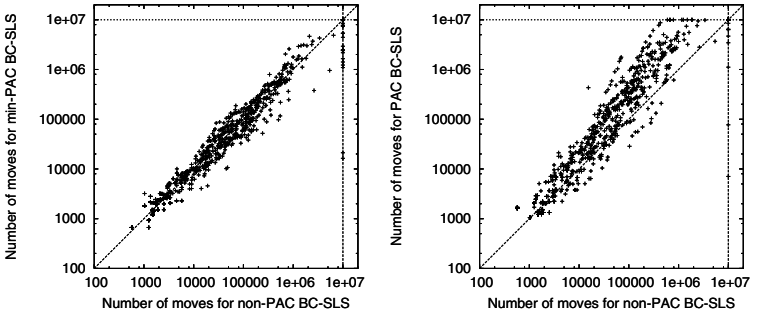


Fig. 4. Non-PAC vs min-PAC BC SLS (left), non-PAC vs PAC BC SLS (right)

3.2 Experiments with Non-PAC and PAC Variant with Fixed Noise Parameter

Before developing adaptive noise mechanisms for BC SLS (Sect. 4), we look at the performance of BC SLS with the fixed noise parameter setting $p = 0.5$. We experiment with a prototype which is a relatively straightforward implementation of BC SLS constructed on top of the bc2cnf Boolean circuit simplifier/CNF translator [20]. In the implementation, only subset minimal justifications are considered for greedy moves. In all the experiments of this paper we use as main benchmarks a set of Boolean circuits encoding the problem of bounded model checking of various asynchronous systems for deadlocks using the encoding in [21] (as listed in Table 1). Although rather easy for current DPLL solvers, these benchmarks are challenging for typical SLS methods. We limit the number of moves (cutoff) for the variants of BC SLS to 10^7, and run each instance 15 times without restarts. When comparing BC SLS to CNF level SLS procedures, we apply exactly the same Boolean circuit level simplification in bc2cnf to the circuits as in our prototype implementation of BC SLS, and then translate the simplified circuit to CNF with the standard "Tseitin-style" translation.

As the first experiment we compare the essentially incomplete ("non-PAC") version where non-greedy moves are only done upwards ($q = 0$) to two PAC variants (as detailed in Section 3.1): in "min-PAC" 1% of non-greedy moves are randomly selected from the set of *minimal* justifications, while in "PAC" 1% of non-greedy moves are randomly selected from the set of *all* justifications (that is, in both cases we set $q = 0.01$ so that the downward non-greedy moves do not become dominating).

It turns out that the variants "non-PAC" and "min-PAC" have quite similar performance (left in Fig. 4) except that "non-PAC" exceeds the cutoff more often. Surprisingly, the "PAC" version, where also non-minimal random justifications are allowed, does not perform as well as the other two variants (right in Fig. 4). With this evidence, we will in all the following experiments apply the "min-PAC" variant of BC SLS.

In the following experiments, we concentrate on evaluating adaptive noise mechanisms for BC SLS, and compare the resulting methods to adaptive clausal SLS methods. We note that a comparison of ("non-PAC") BC SLS using fixed noise parameter setting with WalkSAT is provided in [7] with the results that BC SLS exhibits typically a one-to-four-decade reduction in the number of moves compared to WalkSAT.

4 Adaptive Noise Strategies for BC SLS

Considering CNF level SLS methods for SAT, it has been noticed that SLS performance can vary critically depending on the chosen noise setting [4], and the optimal noise setting can vary from instance to instance and within families of similar instances. The same phenomenon is present also in BC SLS. The average number of moves over 100 runs of BC SLS with different noise parameter settings is shown in Fig. 5 for two different families of increasingly difficult Boolean circuit instances. This observation has led to the development of an *adaptive noise mechanism* for CNF level SLS in the solver AdaptNovelty+ [4], dismissing the requirement of a pre-tuned noise parameter. This idea has been successfully applied in other SLS solvers as well [22]. We now consider strategies for adapting noise in BC SLS.

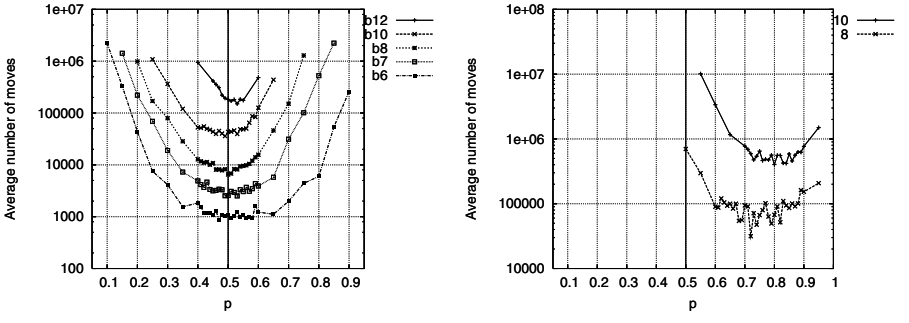


Fig. 5. Average number of moves for BC SLS with different noise parameter settings; left: LTS BMC instance family speed-p, right: factoring instance family braun (see http://www.tcs.tkk.fi/Software/genfacbm/)

4.1 Adaptive Noise in the Context of BC SLS

Following the general idea presented in [4], a generic adaptive noise mechanism for BC SLS is presented as Algorithm 2. Starting from $p = 0$, the noise setting is tuned

Algorithm 2. Generic Adaptive Noise Mechanism

p: noise (initially $p = 0$)
adapt_score: score at latest noise change
adapt_step: step of latest noise change
1: **if** score $<$ adapt_score **then** *%% noise decrease*
2: $p := p - \frac{\phi}{2} \cdot p$
3: adapt_step := step
4: adapt_score := score
5: **else**
6: **if** (step $-$ adapt_step) $>$ WAITINGPERIOD() **then** *%% noise increase*
7: $p := p + \phi \cdot (1 - p)$
8: adapt_step := step
9: adapt_score := score

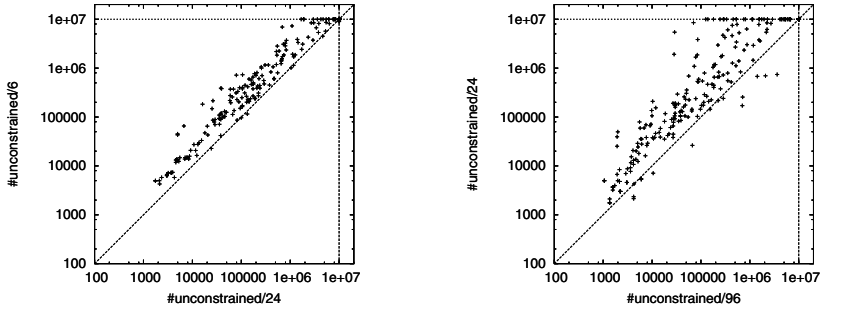


Fig. 6. Comparison of number of moves; left: $\theta = \frac{1}{24}$ vs $\theta = \frac{1}{6}$, right: $\theta = \frac{1}{96}$ vs $\theta = \frac{1}{24}$

during search based on the development of the objective function value. *Every time* the objective function value is improved, noise is decreased according to line 2. If no improvement in the objective function value has been observed during the last WAITINGPERIOD() steps, the noise is increased according to line 7, where $\phi \in]0, 1[$ controls the relative amount of noise increase. Each time the noise setting is changed, the current objective function value is then stored for the next comparison.

Hoos [4] suggests, reporting generally good performance, to use $\phi = \frac{1}{5}$ and the *static* function $\theta \cdot C$ for WAITINGPERIOD(), where $\theta = \frac{1}{6}$ is a constant and C denotes the number of clauses in the CNF instance at hand. These parameter values have been applied also in other CNF level SLS solvers [22].

For BC SLS, as the first step we fix ϕ accordingly to $\frac{1}{5}$, and focus on investigating the effect of applying different waiting periods for noise increases in the context of BC SLS. First we investigate using as WAITINGPERIOD() a static linear function $\theta \cdot U$, where the number U of unconstrained gates is multiplied by a constant θ. In fact, opposed to reported experience with CNF level SLS, it turns out that for BC SLS $\theta = \frac{1}{6}$ is too large: by decreasing θ we can increase the performance of BC SLS. As shown in Fig. 6 (left), by decreasing θ to $\frac{1}{24}$ we witness an evident overall gain in performance against $\theta = \frac{1}{6}$ (left), and again by decreasing θ from $\frac{1}{24}$ to $\frac{1}{96}$ (right).

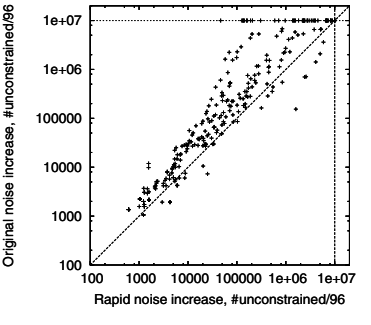


Fig. 7. Comparison of number of moves: rapidly increasing vs original noise mechanism

However, we noticed that changing the overall scheme in the original adaptive noise mechanism leads to even better performance for BC SLS. In the novel scheme, which we call *rapidly increasing*, when the waiting period is exceeded, the noise level is increased after *each* step until we see the first one-step improvement in the objective function. This can be implemented by removing line 8 in Algorithm 2. An example of the resulting improvement is shown in Fig. 7, in which the original and rapidly increasing noise mechanism are compared using $\theta = \frac{1}{96}$. In the following, we will apply the rapidly increasing noise mechanism for BC SLS.

We next compare BC SLS with $\theta = \frac{1}{96}$ to AdaptNovelty+ [23]. Our current prototype of BC SLS does compute the effect of moves on the justification cone and interest set incrementally but is otherwise relatively unoptimized. The results shown in Table 1 are encouraging: BC SLS usually makes much fewer moves and is able to solve more instances in the given time limit than AdaptNovelty+. Although making moves is slower in our BC SLS prototype (around 100000 moves per second on average) than in AdaptNovelty+ (2.5 million per second), BC SLS is very competitive also in running times on these instances as less moves are usually needed for finding a solution.

It it interesting to look at how the noise level fluctuates during a run with different values of θ. An example is provided in Fig. 8 where, using instance

Table 1. Comparison of AdaptNovelty+ and BC SLS (static adaptive noise mechanism, $\theta = \frac{1}{96}$): 101 runs for each instance, 5–minute time limit for each run. succ %: percent of succesful runs.

Instance			BC SLS $\theta = \frac{1}{96}$					AdaptNovelty+				
name	vars	clauses	succ %	time		#moves		succ %	time		#moves	
				min	med	min	med		min	med	min	med
dp_12.fsa-b5-p.bc	953	2966	100	0.1	1.0	4272	149287	100	0.1	0.1	4105	10012
dp_12.fsa-b6-p.bc	1362	4236	100	0.1	0.7	7996	79106	100	0.1	0.1	11006	29010
dp_12.fsa-b7-p.bc	1771	5506	100	0.1	0.6	11504	67705	100	0.1	0.1	23519	72153
dp_12.fsa-b8-p.bc	2180	6776	100	0.2	1.5	21143	142100	100	0.1	0.1	48525	215934
dp_12.fsa-b9-p.bc	2589	8046	100	0.1	4.6	18056	376007	100	0.1	0.3	109929	817996
dp_12.fsa-b5-s.bc	1337	4146	100	0.1	0.1	6234	17642	100	0.1	0.1	9240	22320
dp_12.fsa-b6-s.bc	1746	5416	100	0.1	0.3	9119	37626	100	0.1	0.1	27853	58083
dp_12.fsa-b7-s.bc	2155	6686	100	0.1	1.0	18480	86447	100	0.1	0.1	40098	136157
dp_12.fsa-b8-s.bc	2564	7956	100	0.1	3.1	19857	247490	100	0.1	0.1	60910	369385
dp_12.fsa-b9-s.bc	2973	9226	100	0.3	9.5	38487	730250	100	0.1	2.1	170040	5212785
elevator_1-b4-s.bc	439	1343	100	0.1	0.1	394	1707	100	0.1	0.1	2866	81606
elevator_1-b5-s.bc	698	2149	100	0.1	0.1	1365	3844	100	0.1	0.5	7961	1254582
elevator_1-b6-s.bc	1087	3374	100	0.1	0.8	2507	60052	100	1.4	15.5	3693776	42037729
elevator_2-b6-p.bc	682	2115	100	0.1	0.1	982	4366	100	0.1	5.5	149405	15053510
elevator_2-b7-p.bc	1253	3952	100	0.1	0.7	4120	37607	93	1.3	82.3	3406967	220184348
elevator_2-b6-s.bc	1333	4143	100	0.1	0.2	4389	17761	82	0.3	122.3	832838	329714970
elevator_2-b7-s.bc	2063	6478	100	0.2	1.1	11526	65931	6	36.7	-	94059483	-
elevator_2-b8-s.bc	3123	9919	67	1.7	179.9	79857	7254453	0	-	-	-	-
mmgt_2.fsa-b6-p.bc	654	2036	100	0.1	0.1	569	12878	100	0.1	0.1	11902	308130
mmgt_2.fsa-b7-p.bc	928	2895	100	0.1	0.2	3027	26968	100	0.1	0.3	80656	1468861
mmgt_2.fsa-b8-p.bc	1317	4119	94	0.1	74.3	6293	6395263	100	0.1	34.0	70058	102384691
mmgt_2.fsa-b6-s.bc	1182	3708	100	0.1	0.1	3148	12644	95	1.8	89.2	4798784	239335425
mmgt_2.fsa-b7-s.bc	1723	5429	100	0.1	6.0	8989	347129	0	-	-	-	-
mmgt_2.fsa-b8-s.bc	2381	7530	100	1.2	29.1	60339	1315753	0	-	-	-	-
mmgt_3.fsa-b7-p.bc	1421	4459	100	0.1	0.4	3456	44913	100	0.1	0.1	26370	377011
mmgt_3.fsa-b9-p.bc	2596	8184	100	0.3	29.4	23771	1759402	27	4.8	-	12129665	-
mmgt_3.fsa-b7-s.bc	2588	8226	100	0.2	2.8	11575	154457	0	-	-	-	-
speed_1.fsa-b6-p.bc	498	1514	100	0.1	0.1	385	1159	100	0.1	0.1	1327	26923
speed_1.fsa-b7-p.bc	758	2319	100	0.1	0.1	902	2935	100	0.1	0.1	7364	132024
speed_1.fsa-b8-p.bc	1021	3132	100	0.1	0.1	2125	7914	100	0.1	0.3	43042	919969
speed_1.fsa-b9-p.bc	1284	3944	100	0.1	0.2	3482	17454	100	0.1	2.6	46186	6812540
speed_1.fsa-b10-p.bc	1547	4754	100	0.1	0.4	5382	46156	100	0.4	18.5	1000674	48965683
speed_1.fsa-b12-p.bc	2073	6368	100	0.2	4.8	20250	499851	15	24.3	-	57759838	-
speed_1.fsa-b13-p.bc	2336	7172	100	1.2	40.8	123031	4332369	0	-	-	-	-
speed_1.fsa-b14-p.bc	2599	7974	34	7.0	-	744191	-	0	-	-	-	-
speed_1.fsa-b6-s.bc	666	2026	100	0.1	0.1	603	1278	100	0.1	0.1	2326	13049
speed_1.fsa-b7-s.bc	920	2811	100	0.1	0.1	1238	2409	100	0.1	0.1	6308	47237
speed_1.fsa-b8-s.bc	1175	3596	100	0.1	0.1	2025	4185	100	0.1	0.1	12134	98165
speed_1.fsa-b9-s.bc	1430	4380	100	0.1	0.1	2820	8629	100	0.1	0.1	29602	237623
speed_1.fsa-b10-s.bc	1685	5162	100	0.1	0.2	3514	14860	100	0.1	0.3	52643	790049
speed_1.fsa-b12-s.bc	2195	6722	100	0.1	1.2	8500	100027	100	0.3	6.6	723313	17287780
speed_1.fsa-b13-s.bc	2450	7499	100	0.4	3.7	30637	311209	84	1.5	135.4	3814440	354742108
speed_1.fsa-b14-s.bc	2705	8274	100	0.2	12.3	17063	1072742	15	1.8	-	4647662	-
speed_1.fsa-b15-s.bc	2960	9047	92	1.2	67.9	102953	6013459	3	0.4	-	982942	-

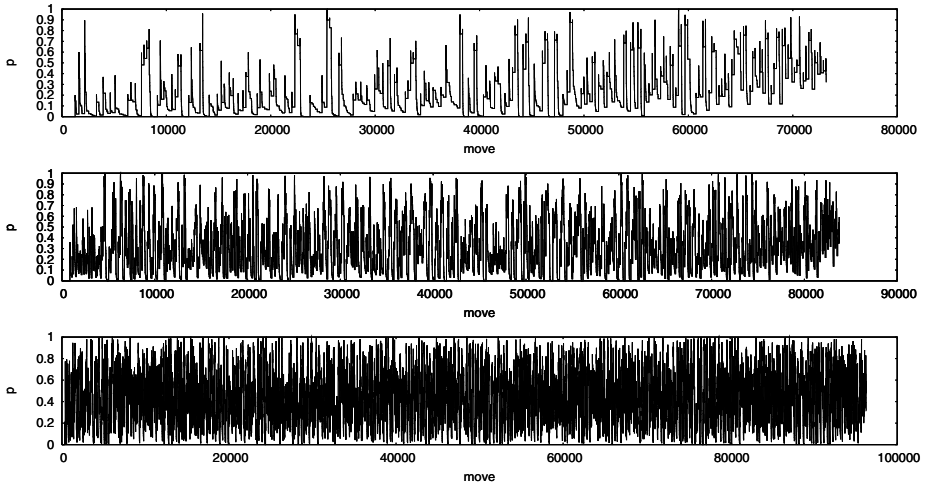


Fig. 8. Noise level fluctuations during a run using $\theta = \frac{1}{6}, \frac{1}{24}, \frac{1}{96}$ (from top to bottom)

`dp_12.fsa-b6-s`, the development of p is shown for $\theta = \frac{1}{6}, \frac{1}{24}, \frac{1}{96}$ (from top to bottom) on runs of similar length. It appears that with larger θ, a significant portion of moves are wasted on plateaus, from which we can escape only with a strong noise increase. On the other hand, for small values, such as $\frac{1}{96}$, the noise level seems to thrash heavily, not focusing on a specific noise range. From another viewpoint, we observed that lowering the value of θ basically raises the average noise level.

Now, the original motivation behind developing adaptive noise mechanisms comes from the fact that the optimal noise level is instance-specific (recall Fig. 5). Apparently a sufficient amount of noise is needed, which can be achieved by lowering the fixed value of θ, but then the hence shortened waiting period for noise increases results in unfocused fluctuations of the noise level. That is, by employing the adaptive noise mechanism based on static waiting periods, we may have only changed the problem of finding the optimal static noise level parameter p into the problem of finding an instance-specific optimal value for θ. This motivates us to consider, opposed to a static waiting period controlled by the addition parameter θ, *dynamic waiting periods* based on the state of search, with the possibility of dismissing the otherwise required constant θ.

We consider two dynamic alternatives: $\text{WAITINGPERIOD}() = \mathsf{jfront}(\mathcal{C}^\alpha, \tau)$ (the size of the current justification frontier), and $\text{WAITINGPERIOD}() = \mathsf{interest}(\mathcal{C}^\alpha, \tau)$ (the size of the current interest set). The intuition behind using front is that since the gate at each step is selected from the justification frontier, the size of the frontier gives us an estimate on the number of possible greedy moves in order to improve the objective function value before increasing the possibility of non-greedy moves (increasing noise). On the other hand, the size of the interest set is precisely the objective function value. Intuitively, the greater the objective function value is, the further we are from a solution, and thus more effort is allowed on finding a good greedy move.

Fig. 9 gives a comparison of performance using the static waiting period with $\theta = \frac{1}{96}$ with the performance resulting from using dynamic waiting period based on frontier

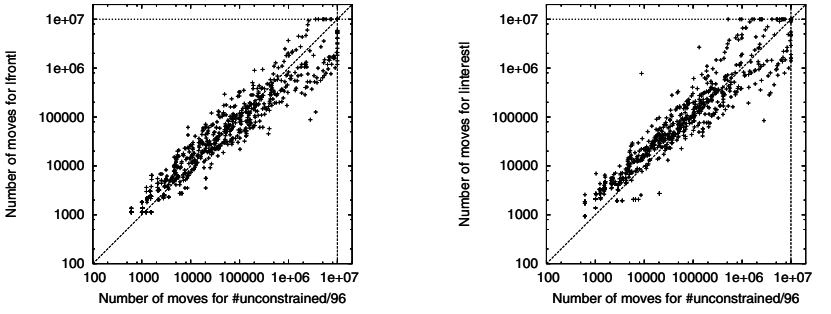


Fig. 9. $\theta = \frac{1}{96}$ vs front (left); $\theta = \frac{1}{96}$ vs interest (right)

size (left) and interest set size (right). The dynamic waiting period results in comparable performance than the static one, although we notice that with the dynamic approach based on frontier size seems to behave more similarly to the static one than the dynamic approach based on interest set size.

This difference is highlighted by looking at the fluctuations of the noise level for the dynamic waiting periods (exemplified in Fig. 10). Especially the noise level fluctuation resulting from the interest set size approach seems to be more focused than when using the static waiting period with $\theta = \frac{1}{96}$ (recall Fig. 8 (bottom)), avoiding some of the observed thrashing behavior without needing to choose a specific value for θ. The question of to what extend thrashing may affect performance is an interesting aspect of further work.

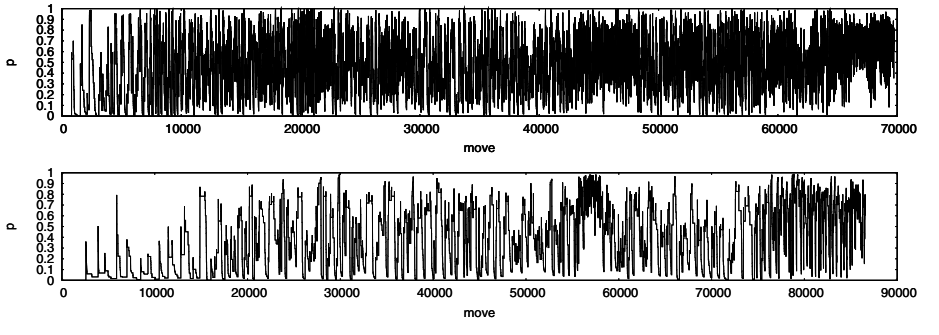


Fig. 10. Noise level fluctuation during a run using front (top) and interest set size (bottom)

5 Conclusions

We extend a recent framework BC SLS [7] for a novel type of stochastic local search (SLS) for SAT. We analyze in detail under which conditions the extended framework is probabilistically approximately complete and under which essentially incomplete. We develop and study adaptive noise mechanisms for BC SLS. The results suggest that, compared to the parameter values for adapting noise used in CNF level SLS methods,

radically different settings are required in BC SLS. As more fundamental changes to the CNF level noise mechanism, we demonstrate improvements in performance for BC SLS by introducing the *rapidly increasing* noise mechanism, and show that there is promise for dismissing the static waiting period constant θ required in current CNF level noise mechanisms by *dynamically* adapting the waiting period for noise increases. Compared to well-known CNF level SLS methods, a prototype implementation of the framework performs favorably w.r.t. the number of moves, showing promise for more optimized implementations of the procedure. An interesting question regarding dynamic waiting periods is whether CNF level SLS methods can gain from similar mechanisms.

Acknowledgements. Research supported by Academy of Finland (grants #122399 and #112016). Järvisalo additionally acknowledges financial support from HeCSE graduate school, Emil Aaltonen Foundation, Jenny and Antti Wihuri Foundation, Nokia Foundation, and Finnish Foundation for Technology Promotion.

References

1. Selman, B., Levesque, H., Mitchell, D.: A new method for solving hard satisfiability problems. In: AAAI, pp. 440–446 (1992)
2. Selman, B., Kautz, H., Cohen, B.: Noise strategies for improving local search. In: AAAI, pp. 337–343 (1994)
3. McAllester, D., Selman, B., Kautz, H.: Evidence for invariants in local search. In: AAAI, pp. 321–326 (1997)
4. Hoos, H.: An adaptive noise mechanism for WalkSAT. In: AAAI, pp. 655–660 (2002)
5. Braunstein, A., Mézard, M., Zecchina, R.: Survey propagation: An algorithm for satisfiability. Random Structures and Algorithms 27(2), 201–226 (2005)
6. Kautz, H., Selman, B.: The state of SAT. Discr. Appl. Math. 155(12), 1514–1524 (2007)
7. Järvisalo, M., Junttila, T., Niemelä, I.: Justification-based non-clausal local search for SAT. In: ECAI. Frontiers in AI and Applications, vol. 178, pp. 535–539. IOS Press, Amsterdam (2008)
8. Junttila, T., Niemelä, I.: Towards an efficient tableau method for Boolean circuit satisfiability checking. In: Palamidessi, C., Moniz Pereira, L., Lloyd, J.W., Dahl, V., Furbach, U., Kerber, M., Lau, K.-K., Sagiv, Y., Stuckey, P.J. (eds.) CL 2000. Lecture Notes in Computer Science (LNAI), vol. 1861, pp. 553–567. Springer, Heidelberg (2000)
9. Kuehlmann, A., Ganai, M., Paruthi, V.: Circuit-based Boolean reasoning. In: DAC, pp. 232–237. ACM, New York (2001)
10. Kuehlmann, A., Paruthi, V., Krohm, F., Ganai, M.K.: Robust Boolean reasoning for equivalence checking and functional property verification. IEEE T-CAD 21(12), 1377–1394 (2002)
11. Thiffault, C., Bacchus, F., Walsh, T.: Solving non-clausal formulas with DPLL search. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 663–678. Springer, Heidelberg (2004)
12. Safarpour, S., Veneris, A., Drechsler, R., Lee, J.: Managing don't cares in Boolean satisfiability. In: DATE 2004. IEEE, Los Alamitos (2004)
13. Sebastiani, R.: Applying GSAT to non-clausal formulas. J. Artif. Intell. Res. 1, 309–314 (1994)
14. Kautz, H., McAllester, D., Selman, B.: Exploiting variable dependency in local search. In: IJCAI poster session (1997), http://www.cs.rochester.edu/u/kautz/papers/dagsat.ps

15. Pham, D., Thornton, J., Sattar, A.: Building structure into local search for SAT. In: IJCAI, pp. 2359–2364 (2007)
16. Hoos, H.H.: On the run-time behaviour of stochastic local search algorithms for SAT. In: AAAI, pp. 661–666 (1999)
17. Patterson, D.J., Kautz, H.: Auto-Walksat: A self-tuning implementation of Walksat. In: SAT, 4th Workshop on Theory and Application of Satisfiability Testing (2001)
18. Selman, B., Kautz, H.: An empirical study of greedy local search for satisfiability testing. In: AAAI, pp. 46–51 (1993)
19. Hirsch, E., Kojevnikov, A.: UnitWalk: A new SAT solver that uses local search guided by unit clause elimination. Ann. Math. Artif. Intell. 43(1), 91–111 (2005)
20. Junttila, T.: The BC package and a file format for constrained Boolean circuits, http://www.tcs.hut.fi/~tjunttil/bcsat/
21. Heljanko, K.: Bounded reachability checking with process semantics. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154, pp. 218–232. Springer, Heidelberg (2001)
22. Li, C., Wei, W., Zhang, H.: Combining adaptive noise and look-ahead in local search for SAT. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 121–133. Springer, Heidelberg (2007)
23. Tompkins, D., Hoos, H.: UBCSAT: An implementation and experimentation environment for SLS algorithms for SAT and MAX-SAT. In: Hoos, H., Mitchell, D.G. (eds.) SAT 2004. LNCS, vol. 3542, pp. 306–320. Springer, Heidelberg (2005)

The Max-Atom Problem and Its Relevance*

Marc Bezem[1], Robert Nieuwenhuis[2], and Enric Rodríguez-Carbonell[2]

[1] Department of Informatics, University of Bergen, Norway
[2] Technical University of Catalonia (UPC), Barcelona, Spain

Abstract. Let F be a conjunction of atoms of the form $\max(x, y)+k \geq z$, where x, y, z are variables and k is a constant value. Here we consider the satisfiability problem of such formulas (e.g., over the integers or rationals). This problem, which appears in unexpected forms in many applications, is easily shown to be in NP. However, decades of efforts (in several research communities, see below) have not produced any polynomial decision procedure nor an NP-hardness result for this -apparently so simple- problem.

Here we develop several ingredients (small-model property and lattice structure of the model class, a polynomially tractable subclass and an inference system) which altogether allow us to prove the existence of small unsatisfiability certificates, and hence membership in NP intersection co-NP. As a by-product, we also obtain a weakly polynomial decision procedure.

We show that the Max-atom problem is PTIME-equivalent to several other well-known -and at first sight unrelated- problems on hypergraphs and on Discrete Event Systems, problems for which the existence of PTIME algorithms is also open. Since there are few interesting problems in NP intersection co-NP that are not known to be polynomial, the Max-atom problem appears to be relevant.

Keywords: constraints, max-plus algebra, hypergraphs.

1 Introduction

Difference Logic (DL) is a well-known fragment of linear arithmetic in which atoms are constraints of the form $x+k \geq y$, where x, y are variables and the *offset* k is a constant value. Due to its many applications to verification (e.g., timed automata), it is one of the most ubiquitous theories in the context of Satisfiability Modulo Theories (SMT). In SMT systems, a *theory solver* is essentially a decision procedure for the satisfiability of *conjunctions* of theory atoms. For DL satisfiability is equivalent to the absence of negative cycles in the digraph having one edge $x \xrightarrow{k} y$ for each atom $x+k \geq y$, and can be decided in polynomial time (e.g., by the Bellman-Ford algorithm; cf. [NOT06] for background on SMT and algorithms for DL, among other theories).

* Partially supported by Spanish Ministry Educ. and Science LogicTools-2 project (TIN2007-68093-C02-01).

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 47–61, 2008.

Motivated by the need of SMT techniques for reasoning about delays in digital circuits, it is natural to extend the atoms of DL to *max-atoms* of the form $\max(x, y)+k \geq z$. The satisfiability of conjunctions of such constraints appears to be a new problem, hereafter referred to as the *Max-atom problem*. The Max-atom problem is easily seen to belong to NP, since after guessing in each atom $\max(x, y)+k \geq z$ which one of x and y is the maximal variable, the problem reduces to DL. As in DL, there is no essential difference here between interpretations over integers or rationals[1]: Given a conjunction of n atoms with rational offsets $\max(x_i, y_i)+p_i/q_i \geq z_i$, for i in $1 \ldots n$, if lcm is the least common multiple of the $q_i's$, one can express each atom as $\max(x_i, y_i)+r_i/lcm \geq z_i$ for certain r_i's and solve the equisatisfiable conjunction of atoms $\max(x_i, y_i)+r_i \geq z_i$ over the integers. Therefore, unless explicitly stated otherwise, here we will only consider integer models and offsets.

The language of conjunctions of max-atoms of the form $\max(x, y)+k \geq z$ is quite expressive, and many interesting problems can be modeled by polynomially many such max-atoms. Some simple examples follow. DL literals $x+k \geq y$ can of course be expressed as $\max(x, x)+k \geq y$. Equalities $\max(x, y)+k = z$ can be written as $\max(x, y)+k \geq z \ \wedge \ z-k \geq x \ \wedge \ z-k \geq y$. Strict inequalities $\max(x, y)+k > z$ can be expressed as $\max(x, y)+k-1 \geq z$. One can express max on both sides, as in $\max(x, y)+k = \max(x', y')+k'$ by introducing a fresh variable z and writing $\max(x, y)+k = z \ \wedge \ \max(x', y')+k' = z$. One can also express different offsets on different arguments of max; for instance $\max(x+5, y-3) \geq z$ can be written as $\max(x, y')+5 \geq z \ \wedge \ y'+8 = y$, where y' is fresh. Furthermore, since $\max(e_1, e_2, e_3)$ is equivalent to $\max(e_1, \max(e_2, e_3))$, one can express nested or larger-arity max-atoms such as $\max(e_1, e_2, e_3) \geq z$ by writing $\max(e_1, x) \geq z \ \wedge \ \max(e_2, e_3) = x$, where x is fresh.

A less simple equivalence (see Section 5) exists with a problem used in Control Theory for modeling Discrete Event Systems. It amounts to solving *two-sided linear max-plus systems:* sets of equations of the form

$$\max(\ x_1+k_1,\ \ldots,\ x_n+k_n\ ) = \max(\ x_1+k'_1,\ \ldots,\ x_n+k'_n\ )$$

where *all* n variables of the system occur on both sides of every equation, which makes it non-trivial to show that max-atoms can be equivalently expressed in this form. Finding a polynomial algorithm for this problem has been open for more than 30 years in the area of max-plus algebras [BZ06]. An elegant algorithm was given and claimed to be polynomial in [BZ06], but unfortunately in [BNRC08] we have given an example on which it behaves exponentially. Currently still no polynomial algorithm is known.

Yet another equivalent problem (see again Section 5) concerns shortest paths in directed weighted hypergraphs. In such hypergraphs, an edge goes from a set of vertices to another vertex. Hence a natural notion of a hyperpath (from a set of vertices to a vertex) is a tree, and a natural notion of length of the hyperpath is the maximal length (the sum of the weights) of a path from a leaf to the root of this tree. For arbitrary directed hypergraphs with positive

[1] Except, possibly, for the weakly polynomial algorithm that will be given in Section 3.

or negative weights, no polynomial algorithm for determining (the existence of) such shortest hyperpaths has been found.

Slight increases in expressive power lead to NP-hardness. For instance, having both max and min it is easy to express any 3-SAT problem with variables $x_1 \dots x_n$, by: (i) an atom $T > F$ (T, F are variables); (ii) for all x_i the atoms $\min(x_i, x_i') = F$ and $\max(x_i, x_i') = T$; (iii) for each clause like $x_p \vee \overline{x_q} \vee x_r$, an atom $\max(x_p, x_q', x_r) \geq T$.

Altogether, decades of efforts in the hypergraph and the max-plus communities have not produced any polynomial decision procedure nor an NP-hardness result for the different versions of the –apparently so simple– Max-atom problem. In this paper we give several interesting new insights.

In Section 2 we first prove some relevant results on the models of sets (conjunctions) of max-atoms: we give a small-model property, and show that the model class is a (join semi-) lattice. These properties allow us to prove that a set of max-atoms is unsatisfiable if, and only if, it has an unsatisfiable subset which is *right-distinct*, i.e., where each variable occurs at most once as a right-hand side of a max-atom.

In Section 3 we define *max-derivations* as transformation systems on states (assignments to the variables) as a formalism for searching models, and use the properties of the previous section to obtain a weakly polynomial algorithm for the integers, which is also a strongly polynomial one for a relevant subclass of problems.

In Section 4 we define a *chaining* inference system for max-atoms of the form $\max(x_1 + k_1, \dots, x_m + k_m) \geq z$, and building upon the previous results we show that it is sound and refutation complete. Moreover, we prove that for right-distinct sets chaining can be turned into a polynomial-time decision procedure, thus showing that the Max-atom problem is in co-NP (one only needs to guess the small unsatisfiability certificate: the right-distinct unsatisfiable subset). Since there are few interesting problems in NP ∩ co-NP that are not known to be polynomial, this one appears to be relevant. Moreover, given the history of problems in this class, such as deciding primality [AKS04], there is hope for a polynomial-time algorithm.

The paper ends with the proofs of equivalence with solving two-sided linear max-plus systems and shortest paths in hypergraphs (Section 5) and the conclusions (Section 6).

2 Models of Conjunctions of Max-Atoms

The following lemma ensures that models of a set of max-atoms are invariant under "uniform" translations:

Lemma 1. *Given a set of max-atoms S defined over the variables V and an assignment $\alpha : V \to \mathbb{Z}$ which is a model of S, for any $d \in \mathbb{Z}$ the assignment α' defined by $\alpha'(x) = \alpha(x) + d$ is a model of S.*

Definition 1. *Given a set of variables V, the* size *of an assignment $\alpha : V \to \mathbb{Z}$ is the difference between the largest and the smallest value assigned to the variables, i.e.,* $\text{size}(\alpha) = \max_{x,y \in V}(\alpha(x) - \alpha(y))$.

Lemma 2 (Small Model Property). *If a set of max-atoms S is satisfiable, then it has a model of size at most the sum of the absolute values of the offsets, i.e., at most*

$$K_S = \sum_{\max(x,y)+k \geq z \ \in \ S} |k|.$$

Proof. We may assume that all constraints in the set are equations: replace each max-atom $\max(x, y) + k \geq z$ by the constraints $\max(x, y) + k = z'$ and $\max(z, z') = z'$. The class of models does not change essentially by adding these auxiliary constraints and variables, as one just has to add/omit interpretations for the fresh variables. Furthermore, the sum of the absolute values of the offsets does not change. Therefore, we may assume that S is a set of constraints of the form $\max(x, y) + k = z$ (where possibly x and y are the same variable).

Let α be a model of S. Based on α we define a weighted graph whose vertices are the variables. For every constraint $\max(x, y) + k = z$, if $\alpha(x) \geq \alpha(y)$ then we add a red edge (x, z) with weight k and a green edge (y, x) without a weight; and otherwise, if $\alpha(y) > \alpha(x)$ then we add a red edge (y, z) with weight k and a green edge (x, y) without a weight. While changing the model, the graph will remain all the time the same.

A red (weakly) connected component is a subgraph such that there are red paths between any two variables in the subgraph, where the red edges may be used in any direction. The *segment* of a red connected component is the range of integers from the lowest value to the highest one assigned to the variables in the component. The size of such a segment is at most the sum of the absolute values of the weights of the edges in the component.

Red connected components partition the set of variables. If their segments overlap, then already $\text{size}(\alpha) \leq K_S$. If there is a gap somewhere, say of size p, then the gap is closed by a suitable translation, e.g., by decreasing by p all values assigned to variables above the gap. This respects all red edges and their weights since the gap is between segments of red connected components and components are translated as a whole. Green edges are also respected since we only close gaps and never a variable x with initially a higher value than another variable y ends up with a value strictly lower than y. Since all edges are respected we keep a model, all the time closing gaps until there are no gaps left. We end up with a model α' without gaps and hence $\text{size}(\alpha') \leq K_S$. □

The previous lemma gives an alternative proof of membership in NP of the Max-atom problem: it suffices to guess a "small" assignment; checking that it is indeed a model is trivially in P.

Lemma 3 below proves that the model class of a set of max-atoms is a (join semi-) lattice, where the partial ordering is $\geq$ (pointwise $\geq$):

Definition 2. *Given a set of variables V and assignments $\alpha_1, \alpha_2 : V \to \mathbb{Z}$, we write $\alpha_1 \geq \alpha_2$ if for all $x \in V$, $\alpha_1(x) \geq \alpha_2(x)$.*

Definition 3. *Given a set of variables V and two assignments $\alpha_1, \alpha_2 : V \to \mathbb{Z}$, the* supremum *of α_1 and α_2, denoted by $\sup(\alpha_1, \alpha_2)$, is the assignment defined by $\sup(\alpha_1, \alpha_2)(x) = \max(\alpha_1(x), \alpha_2(x))$ for all $x \in V$.*

Lemma 3. *Given a set of max-atoms S defined over the variables V and two assignments $\alpha_1, \alpha_2 : V \to \mathbb{Z}$, if $\alpha_1 \models S$ and $\alpha_2 \models S$ then $\sup(\alpha_1, \alpha_2) \models S$.*

Proof. Let us denote $\sup(\alpha_1, \alpha_2)$ by α^*. Assume $\alpha_1 \models S$ and $\alpha_2 \models S$ and let $\max(x, y)+k \geq z$ be an atom in S. By assumption, $\max(\alpha_i(x), \alpha_i(y))+k \geq \alpha_i(z)$ for $i = 1, 2$. Also by definition for $i = 1, 2$ we have $\alpha^*(x) \geq \alpha_i(x)$ and $\alpha^*(y) \geq \alpha_i(y)$, so $\max(\alpha^*(x), \alpha^*(y)) + k \geq \alpha_i(z)$. Thus $\max(\alpha^*(x), \alpha^*(y)) + k \geq \alpha^*(z)$, that is, the atom $\max(x, y) + k \geq z$ is satisfied by α^*. Hence $\alpha^* \models S$. □

Using the previous lemmas, we have the following result:

Lemma 4. *Let S be a set of max-atoms, and let z be a variable such that for some $r > 1$ all max-atoms with z as right-hand side are $L_1, ..., L_r$. The set S is satisfiable if and only if all $S - \{L_i\}$ (i in $1 \dots r$) are satisfiable.*

Proof. The "only if" implication is trivial, since $S - \{L_i\} \subseteq S$ for all i in $1 \dots r$. Now, for the "if" implication, let α_i be a model of $S - \{L_i\}$. By Lemma 1, for every i in $2 \dots r$ we can assume w.l.o.g. that $\alpha_i(z) = \alpha_1(z)$. Let us define $\alpha^* = \sup(\alpha_1, ..., \sup(\alpha_{r-1}, \alpha_r)...)$. Then $\alpha^*(z) = \alpha_i(z)$ for all i in $1 \dots r$. Moreover, since for all i in $1 \dots r$ we have in particular $\alpha_i \models S - \{L_1, ..., L_r\}$, by iterating Lemma 3, $\alpha^* \models S - \{L_1, ..., L_r\}$. It remains to be seen that $\alpha^* \models L_i$ for any i in $1 \dots r$. Let thus $\max(x, y) + k \geq z$ be L_i, for a given i in $1 \dots r$. Since $r > 1$, there is j in $1 \dots r$ such that $i \neq j$. Since $\alpha_j \models S - \{L_j\}$ and $i \neq j$, $\alpha_j \models L_i$. So $\max(\alpha^*(x), \alpha^*(y)) + k \geq \max(\alpha_j(x), \alpha_j(y)) + k \geq \alpha_j(z) = \alpha^*(z)$. Hence $\alpha^* \models L_i$. □

The next definition and lemma will be paramount for building short certificates of unsatisfiability:

Definition 4. *A set of max-atoms S is said to be* right-distinct *if variables occur at most once as right-hand sides, i.e., for every two distinct max-atoms $\max(x, y) + k \geq z$ and $\max(x', y') + k \geq z'$ in S we have $z \neq z'$.*

Lemma 5. *Let S be a set of max-atoms. If S is unsatisfiable, then there exists an unsatisfiable right-distinct subset $S' \subseteq S$.*

Proof. Let V be the set of variables over which S is defined. Let us prove the result by induction on $N = |S| - |\{z \in V \mid z \text{ appears as a right-hand side in } S\}|$:

- Base step: $N = 0$. Then all variables appearing as right-hand sides are different. So S is right-distinct, and we can take $S' = S$.
- Inductive step: $N > 0$. Then there is a variable which appears at least twice as a right-hand side. Let z be such a variable. By Lemma 4, since S is unsatisfiable, there exists an atom $L \in S$ with right-hand side z such that $S - \{L\}$ is unsatisfiable. Now, by induction hypothesis on $S - \{L\}$ there is an unsatisfiable right-distinct set $S' \subseteq S - \{L\} \subset S$. □

3 Max-Derivations

W.l.o.g. in this section max-atoms are of the form $\max(x, y) + k \geq z$ with $x \neq z$, $y \neq z$. This can be assumed by removing trivial contradictions $\max(x, x) + k \geq x$ $(k < 0)$, trivial tautologies $\max(x, y) + k \geq x$ $(k \geq 0)$, and by replacing $\max(x, y) + k \geq x$ by $\max(y, y) + k \geq x$ if $k < 0$ and $x \neq y$.

Definition 5. *Given a set of max-atoms S defined over the variables V and two assignments α, α', we write $\alpha \rightarrow_S \alpha'$ (or simply $\alpha \rightarrow \alpha'$, if S is understood from the context) if there is a max-atom* $\max(x, y) + k \geq z \in S$ *such that:*

1. $\alpha'(z) = \max(\alpha(x), \alpha(y)) + k$
2. $\alpha'(z) < \alpha(z)$ *(hence we say that z* decreases *in this step)*
3. $\alpha'(u) = \alpha(u)$ *for all $u \in V$, $u \neq z$.*

Any sequence of steps $\alpha_0 \rightarrow \alpha_1 \rightarrow \cdots$ is called a max-derivation *for S.*

Lemma 6. *Let S be a set of max-atoms defined over the variables V. An assignment $\alpha : V \rightarrow \mathbb{Z}$ is a model for S if and only if α is* final*, i.e., there is no α' such that $\alpha \rightarrow \alpha'$.*

The following lemma expresses that max-derivations, while decreasing variables, never "break through" any model:

Lemma 7. *Let S be a set of max-atoms and let α be a model of S. If $\alpha_0 \rightarrow \cdots \rightarrow \alpha_m$ and $\alpha_0 \geq \alpha$, then $\alpha_m \geq \alpha$.*

Proof. By induction over m, the length of the derivation. For $m = 0$ there is nothing to prove. Now, if $m > 0$ the step $\alpha_0 \rightarrow \alpha_1$ is by an atom $\max(x, y) + k \geq z$. Let us prove that $\alpha_1 \geq \alpha$. We only need to show that the inequality holds for the variable that changes, which is z; and indeed $\alpha_1(z) = \max(\alpha_0(x), \alpha_0(y)) + k \geq \max(\alpha(x), \alpha(y)) + k \geq \alpha(z)$. Now, by induction hypothesis $\alpha_m \geq \alpha$. □

The previous lemma, together with the Small Model Property (Lemma 2), provides us with a weakly polynomial algorithm, i.e., an algorithm whose runtime is polynomial in the input size if numbers are encoded in unary.

Theorem 1. *The Max-atom problem over the integers is weakly polynomial.*

Proof. Let S be a conjunction of max-atoms, with variables V, where $|V| = n$. For deciding the satisfiability of S one can construct an arbitrary max-derivation, starting, e.g., from the assignment α_0 with $\alpha_0(x) = 0$ for all x in V. At each step, one variable decreases by at least one. If S is satisfiable, by the Small Model Property and by Lemma 1, there is a model α such that $-K_S \leq \alpha(x) \leq 0$ for all x in V. Moreover, by the previous lemma, no variable x will ever get lower than $\alpha(x)$ in the derivation. Altogether this means that, if no model is found after $n \cdot K_S$ steps, then S is unsatisfiable. □

Note that the previous result does not directly extend to the case of the rationals since the transformation described in the introduction may produce an exponential blow-up in the *value* of the offsets.

As a corollary of the proof of the previous theorem, we obtain a PTIME decision procedure for sets of atoms of the forms $\max(x, y) \geq z$ or $\max(x, y) > z$. More generally, this also applies to *K-bounded sets*, where in S the absolute values of all offsets are bounded by a given constant K.

Example 1. Let S be the following set of max-atoms:

$$S = \{u - 10 \geq x, \quad z \geq y, \quad \max(x, y) - 1 \geq z, \quad \max(x, u) + 25 \geq z\},$$

and let α_0 be the assignment with $\alpha_0(x) = \alpha_0(y) = \alpha_0(z) = \alpha_0(u) = 0$. This initial assignment α_0 violates $u - 10 \geq x$, which allows us to decrease x and assign it the value -10: in terms of max-derivations $\alpha_0 \rightarrow \alpha_1$, where α_1 is the assignment with $\alpha_1(x) = -10$, $\alpha_1(y) = \alpha_1(z) = \alpha_1(u) = 0$.

Now the assignment α_1 only violates $\max(x, y) - 1 \geq z$, which forces z to take the value -1: in terms of max-derivations, $\alpha_1 \rightarrow \alpha_2$, where α_2 is the assignment with $\alpha_2(x) = -10$, $\alpha_2(y) = 0$, $\alpha_2(z) = -1$, $\alpha_2(u) = 0$. Then α_2 only violates $z \geq y$, which forces y to take the value -1 too: $\alpha_2 \rightarrow \alpha_3$, where α_3 is the assignment with $\alpha_3(x) = -10$, $\alpha_3(y) = \alpha_3(z) = -1$, $\alpha_3(u) = 0$.

It is easy to see that 11 iterations of each of the last two steps will be needed to find a model: finally we will have a derivation $\alpha_0 \rightarrow^* \alpha$ with $\alpha(x) = -10$, $\alpha(y) = \alpha(z) = -11$, $\alpha(u) = 0$; since there is no α' such that $\alpha \rightarrow \alpha'$, α is a model of S, hence S is satisfiable.

Notice that, if we replace 10 in S by larger powers of 10, we get a family of inputs whose sizes increase linearly, but for which the number of steps of the max-derivations reaching to a model grows exponentially. Since the number of steps is polynomial in the *value* of the offsets, and not in the *sizes* of the offsets, the algorithm based on max-derivations can be *weakly* polynomial but not polynomial.

Now, if we consider the set of max-atoms $S' = S \cup \{\max(x, y) + 9 \geq u\}$, we note that α above does not satisfy the new constraint. So we can decrease u and assign it the value -1, which makes $u - 10 \geq x$ false and forces x to take the value -11. Then $\max(x, y) - 1 \geq z$ is violated, and z is decreased to -12. Finally $z \geq y$ becomes false, so y is assigned -12. The loop of these four steps can be repeated over and over, making all variables decrease indefinitely. Thus, S' is unsatisfiable as no model is found within the bound of $n \cdot K_S$ steps given in the previous theorem.

4 Chaining Inference System and Membership in Co-NP

In this section we deal with the (equivalent in expressive power) language of max-atoms of the form $\max(x_1+k_1, ..., x_n+k_n) \geq z$. Here, T always stands for a max-expression of the form $\max(y_1+k'_1, \ldots, y_m+k'_m)$ with $m \geq 0$; when written inside a max-expression the whole expression is considered flattened, so then $\max(T, z+k)$ represents $\max(\ y_1+k'_1, \ldots, y_m+k'_m,\ z+k\ )$.

Definition 6. *The* Max-chaining *inference rule is the following:*

$$\frac{\max(x_1{+}k_1, ..., x_n{+}k_n) \geq y \qquad \max(T, y{+}k) \geq z}{\max(T, x_1{+}k_1{+}k, \ldots, x_n{+}k_n{+}k) \geq z} \qquad \textit{(Max-chaining)}$$

Definition 7. *The* Max-atom simplification rules *are as follows:*

$$\frac{\max(T, x{+}k) \geq x}{\max(T) \geq x} \qquad \textit{if } k < 0 \qquad \textit{(Max-atom simplification-1)}$$

$$\frac{\max(T, x{+}k, x{+}k') \geq y}{\max(T, x{+}k') \geq y} \qquad \textit{if } k \leq k' \qquad \textit{(Max-atom simplification-2)}$$

Theorem 2. *The Max-chaining rule and the Max-atom simplification rules are sound, i.e., the conclusions of the inference rules are logical consequences of their respective premises. Moreover, for each one of the Max-atom simplification rules, the conclusion and the premise are logically equivalent.*

Theorem 3. *Max-chaining, together with the Max-atom simplifications rules, is refutation complete. That is, if S is an unsatisfiable set of max-atoms that is closed under the Max-chaining and Max-atom simplification rules, then there is a* contradiction *in S, i.e., a max-atom of the form* $\max() \geq x$.

Proof. We prove a slightly stronger result, namely the refutation completeness with a concrete *ordered* application strategy, assuming an ordering on the variables $x_1 > \ldots > x_n$ occurring in S. We prove that if there is no contradiction in S then S is satisfiable. This is done by induction on n.
Base case: if $n = 1$ all atoms in S are of the form $\max(x{+}k_1, \ldots, x{+}k_m) \geq x$, with $m \geq 1$, and where at least one of the k_i is positive (otherwise *Max-atom simplification-1* generates the contradiction $\max() \geq x$). Therefore these max-atoms are tautologies and hence satisfiable.
Induction step. Assume $n > 1$. Let S_1 be the subset of S of its max-atoms in which the variable x_1 occurs. Let SR_1 and SL_1 be the subsets of S_1 of max-atoms in which x_1 occurs exactly once, only at the right-hand sides and only at the left-hand sides, respectively. By an easy induction applying the previous theorem, all max-atoms in S_1 are logical consequences of the ones in SR_1 and SL_1, since S is closed under the Max-Simplification rules. Let S_1' be the set of the $|SR_1|{\cdot}|SL_1|$ max-atoms that can be obtained by applying the max-chaining steps on x_1 between max-atoms of these two sets. Now let S_2 be the set $S \setminus S_1$. Note that it is closed under the Max-chaining and Max-atom simplification rules and that $S_2 \supseteq S_1'$. Since S_2 has one variable (x_1) less than S, by induction hypothesis there exists a model α for S_2.

We will now *extend* α to a model α' for S. That is, we will have $\alpha'(x_i) = \alpha(x_i)$ for all $i > 1$, and in addition α' will also be defined for x_1, in such a way that $\alpha' \models SR_1 \cup SL_1$, which implies $\alpha' \models S_1$, and hence, since $\alpha \models S_2$, we will obtain $\alpha' \models S$.

Let SR_1 be of the form $\{ T_1 \geq x_1, \ldots, T_m \geq x_1 \}$ $(m > 0)$, and let $\alpha(T)$ denote the evaluation of T under the assignment α.[2] Now we define $\alpha'(x_1)$ to be $\min(\alpha(T_1), \ldots, \alpha(T_m))$. W.l.o.g., say, $\alpha'(x_1) = \alpha(T_1)$. Let T_1 be of the form $\max(y_1 + k_1, \ldots, y_m + k_m)$, so that $\alpha'(x_1) = \max(\alpha(y_1) + k_1, \ldots, \alpha(y_m) + k_m)$. Clearly α' satisfies by construction all atoms in SR_1. It only remains to show that α' is also a model of SL_1, i.e., of the atoms of the form $\max(x_1 + k, T) \geq z$. For each such atom in SL_1, the corresponding conclusion by max-chaining with the atom $\max(y_1 + k_1, \ldots, y_m + k_m) \geq x_1$ is the atom $\max(y_1 + k_1 + k, \ldots, y_m + k_m + k, \ T\ ) \geq z$, which is in S_2 and is hence satisfied by α. So, as $\alpha'(x_1) = \max(\alpha(y_1) + k_1, \ldots, \alpha(y_m) + k_m)$, also $\max(x_1 + k, T) \geq z$ is satisfied by α'. □

Notice that the algorithm described in the proof of the previous theorem is a generalization of the Fourier-Motzkin elimination procedure.

Example 2. Let us consider again the system introduced in Example 1 extended with $\max(x, y) + 9 \geq u$, which makes it unsatisfiable. Atoms are written now in the format used in this section.

$$\begin{array}{llll} \{ \max(u - 10) & \geq x, & \max(z) & \geq y, \\ \ \max(x - 1, y - 1) & \geq z, & \max(x + 25, u + 25) & \geq z \\ \ \max(x + 9, y + 9) \geq u \} & & & \end{array}$$

By applying a closure strategy as described in the proof of the previous theorem, we get a contradiction:

Rule	**Set of Max-Atoms**
	$\max(u - 10) \geq x$, $\max(z) \geq y$, $\max(x - 1, y - 1) \geq z$, $\max(x + 25, u + 25) \geq z$ $\max(x + 9, y + 9) \geq u$
max-chaining x	$\max(z) \geq y$, $\max(u - 11, y - 1) \geq z$, $\max(u + 15, u + 25) \geq z, \max(u - 1, y + 9) \geq u$
atom-simplification-2	$\max(z) \geq y$, $\max(u - 11, y - 1) \geq z$, $\max(u + 25) \geq z$, $\max(u - 1, y + 9) \geq u$
atom-simplification-1	$\max(z) \geq y$, $\max(u - 11, y - 1) \geq z$, $\max(u + 25) \geq z$, $\max(y + 9) \geq u$
max-chaining y	$\max(u - 11, z - 1) \geq z$, $\max(u + 25) \geq z$, $\max(z + 9) \geq u$
atom-simplification-1	$\max(u - 11) \geq z$, $\max(u + 25) \geq z$, $\max(z + 9) \geq u$
max-chaining z	$\max(u - 2) \geq u$, $\max(u + 34) \geq u$
atom-simplification-1	$\max() \geq u$

Theorem 4. *The Max-atom problem for right-distinct sets is decidable in polynomial time.*

[2] Note that when $SR_1 = \emptyset$, if SL_1 has the form $\{ \max(x_1 + k_1, T_1) \geq z_1, \ldots, \max(x_1 + k_n, T_n) \geq z_n \}$ $(n > 0)$ one just needs to define $\alpha'(x_1) = \max(\alpha(z_1) - k_1, \ldots, \alpha(z_n) - k_n)$. If $SL_1 = \emptyset$ too, then $\alpha'(x_1)$ can be defined arbitrarily.

Proof. For right-distinct sets, the closure process eliminating variables one by one, as explained in the refutation completeness proof, can be done in polynomial time if the Max-atom simplification rules are applied eagerly. The proof shows that after each Max-atom simplification step, its premise can be ignored (i.e., removed) once the conclusion has been added, and that tautologies of the form $\max(\ldots, x+k, \ldots) \geq x$ with $k \geq 0$ can also be ignored. Eliminating one variable x can then be done in polynomial time, since there is only one leftmost premise of chaining with x. After eliminating x, a new right-distinct set of max-atoms with one variable less and at least one atom less is obtained, in which each atom has arity bounded by the number of variables and the size of the offsets is bounded by the sum of the sizes of the offsets in the input. □

Example 3. In the previous example, an unsatisfiable right-distinct subset is:

$$\{\ \max(u-10) \geq x,\ \ \max(z) \geq y,\ \ \max(x-1, y-1) \geq z,\ \ \max(x+9, y+9) \geq u\ \}.$$

Applying the polynomial-time closure we get a contradiction:

Rule	**Set of Max-Atoms**
	$\max(u-10) \geq x$ $\max(z) \geq y$ $\max(x-1, y-1) \geq z$ $\max(x+9, y+9) \geq u$
max-chaining x	$\max(z) \geq y$ $\max(u-11, y-1) \geq z$ $\max(u-1, y+9) \geq u$
atom-simplification-1	$\max(z) \geq y$ $\max(u-11, y-1) \geq z$ $\max(y+9) \geq u$
max-chaining y	$\max(u-11, z-1) \geq z$ $\max(z+9) \geq u$
atom-simplification-1	$\max(u-11) \geq z$ $\max(z+9) \geq u$
max-chaining z	$\max(u-2) \geq u$
atom-simplification-1	$\max() \geq u$

Theorem 5. *The Max-atom problem is in co-NP.*

Proof. By Lemma 5, if a set of max-atoms is unsatisfiable, it has a right-distinct unsatisfiable subset. This subset is a small unsatisfiability certificate, which, by the previous theorem, can be verified in polynomial time. □

Since there are few interesting problems in NP ∩ co-NP that are not known to be polynomial, this problem (in its several equivalent forms) appears to be relevant. Moreover, given the history of problems in this class, such as deciding primality [AKS04], there is hope for a polynomial-time algorithm.

5 PTIME Equivalences

In this section we show the polynomial reducibility between the Max-atom problem, the satisfiability problem for two-sided linear max-plus systems, and the existence problem of shortest hyperpaths in hypergraphs.

Theorem 6. *The Max-atom problem and the problem of satisfiability of a two-sided linear max-plus system are polynomially reducible to each other.*

Proof. Reducing this kind of max-equations to max-atoms can be done as explained in the introduction. For the reverse reduction, by the Small Model Property, if S is satisfiable then it has a model α such that $\text{size}(\alpha) \leq K_S$ (notice that K_S can be computed in polynomial time). Let $V = \{x_1, \ldots, x_n\}$ be the set of variables over which S is defined. Now, for each variable x_i, we consider the equation

$$\begin{aligned} &\max(\ x_1 - 1, ..., x_{i-1} - 1, x_i + K_S, x_{i+1} - 1, ..., x_n - 1) = \\ &\max(\ x_1, \quad ..., x_{i-1}, \quad x_i + K_S, x_{i+1}, \quad ..., x_n), \end{aligned}$$

which is equivalent to $x_i + K_S \geq x_j$, i.e., $K_S \geq x_j - x_i$ for all j in $1 \ldots n$, $j \neq i$. Let S'_0 be the two-sided linear max-plus system consisting of these n equations. Now we add new equations to S'_0 to obtain a system S' which is equisatisfiable to S. This is achieved by replacing every max-atom $\max(x_{i_1}, x_{i_2}) + k \geq x_{i_3}$ in S by the equation

$$\begin{aligned} &\max(\ x_{i_1} + k, x_{i_2} + k, x_{i_3}, \quad x_j - K_S - |k| - 1, ...) = \\ &\max(\ x_{i_1} + k, x_{i_2} + k, x_{i_3} - 1, x_j - K_S - |k| - 1, ...), \end{aligned}$$

where j ranges over all variable indices different from i_1, i_2, i_3 (if any of the indices i_1, i_2 or i_3 coincide, an obvious simplification must be applied). The offset $-K_S - |k| - 1$ has been chosen so that variables with this offset do not play a role in the maxima. If we leave them out, it is clear that the resulting constraint $\max(x_{i_1} + k, x_{i_2} + k, x_{i_3}) = \max(x_{i_1} + k, x_{i_2} + k, x_{i_3} - 1)$ is equivalent to the max-atom $\max(x_{i_1}, x_{i_2}) + k \geq x_{i_3}$. □

For the relationship with shortest hyperpaths, first some preliminary notions on hypergraphs are presented. We do this by contrasting them with the analogous concepts for graphs.

A (directed, weighted) *graph* is a tuple $G = (V, E, W)$ where V is the set of *vertices*, E is the set of *edges* and $W : E \to \mathbb{Z}$ is the *weight function*. Each edge is a pair (s, t) from a vertex $s \in V$ called the *source vertex* to a vertex $t \in V$ called the *target vertex*.

A (directed, weighted) *hypergraph* is a tuple $H = (V, E, W)$ where V is the set of *vertices*, E is the set of *hyperedges* and $W : E \to \mathbb{Z}$ is the *weight function*. Each hyperedge is a pair (S, t) from a non-empty finite subset of vertices $S \subseteq V$ called the *source set* to a vertex $t \in V$ called the *target vertex*. Thus, a graph is a hypergraph in which for all hyperedges the source set consists of a single element.

Given a graph $G = (V, E, W)$ and vertices $x, y \in V$, a *path from x to y* is a sequence of edges defined recursively as follows: (i) if $y = x$, then the empty sequence $\emptyset$ is a path from x to y; (ii) if there is an edge $(z, y) \in E$ and a path $s_{x,z}$ from x to z, then the sequence $s_{x,y}$ obtained by appending (z, y) to the sequence $s_{x,z}$ is a path from x to y.

Given a hypergraph $H = (V, E, W)$, a subset of vertices $X \subseteq V$, $X \neq \emptyset$ and $y \in V$, a *hyperpath from X to y* is a tree defined recursively as follows: (i) if $y \in X$, then the empty tree $\emptyset$ is a hyperpath from X to y; (ii) if there is a hyperedge $(Z, y) \in E$ and hyperpaths t_{X,z_i} from X to z_i for each $z_i \in Z$, then the tree $t_{X,y}$ with root (Z, y) and children the trees t_{X,z_i} for each vertex $z_i \in Z$, is a hyperpath from X to y. Therefore, when viewing graphs as hypergraphs, a path is just a hyperpath where the tree has degenerated into a sequence of edges. This notion of hyperpath corresponds to the *unfolded hyperpaths* or *hyperpath trees* of [AIN92].

Using the weight function W on the edges E of a graph, one can extend the notion of weight to paths. Namely, the *weight of a path p*, denoted by $\omega(p)$, can be defined naturally as follows: (i) if p is $\emptyset$, then $\omega(p) = 0$; (ii) if p is the result of appending the edge e to the path q, then $\omega(p) = W(e) + \omega(q)$.

On the other hand, in the case of hypergraphs several notions of hyperpath weight have been studied [AIN92]. In this paper we consider the one of *rank* (also called the *distance* [GLPN93]) of a hyperpath p, which is defined as: (i) if p is $\emptyset$, then $\omega(p) = 0$; (ii) if p is a tree with root the hyperedge e and children $p_1, ..., p_m$, then $\omega(p) = W(e) + \max\big(\omega(p_1), \ldots, \omega(p_m)\big)$. This natural notion intuitively corresponds to the heaviest path in the tree.

From now on, we will assume that hypergraphs are *finite*, i.e., the set of vertices V is finite.

Example 4. Fig. 1 (a) shows an example of a hypergraph. E.g., the hyperedge $(\{u\}, x)$ has weight -10, while the weight of the hyperedge $(\{u, x\}, z)$ is 25 . The empty tree is a hyperpath from $\{u, y\}$ to y with rank 0; Fig. 1 (b) shows another hyperpath from $\{u, y\}$ to y, with rank 24.

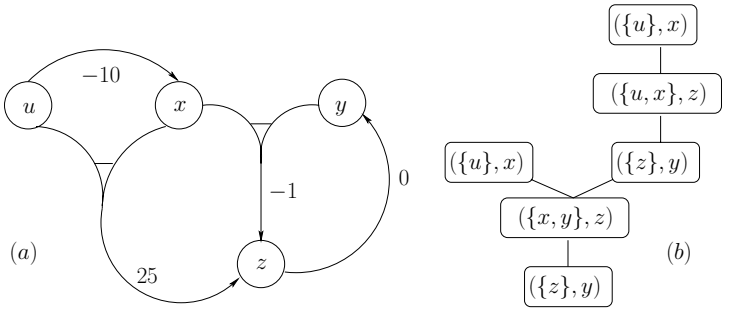


Fig. 1. Example of hypergraph

We now show that the Max-atom problem and the problem of existence of shortest hyperpaths (i.e., with the least rank) in hypergraphs are equivalent, in the sense that they can polynomially be reduced to one another.

Definition 8. *Let $H = (V, E, W)$ be a hypergraph. Given a subset of vertices $X \subseteq V$, $X \neq \emptyset$, the* distance *function $\delta_X : V \to \mathbb{Z} \cup \{\pm\infty\}$ is defined as*

$$\delta_X(y) = \inf\{\omega(t_{X,y}) \mid t_{X,y} \text{ is a hyperpath from } X \text{ to } y\},$$

where for $S \subseteq \mathbb{R}$, we denote by $\inf(S) \in \mathbb{R} \cup \{\pm\infty\}$ the infimum of S.

The distance function δ_X is said to be well-defined *if $\delta_X(y) > -\infty$ for all vertices $y \in V$.*

With this definition, intuitively $+\infty$ means "no hyperpath" and $-\infty$ is related to negative cycles, for instance in the presence of an hyperedge such as $W(\{x\}, x) = -1$.

Our goal is to show that the satisfiability of sets of max-atoms is equivalent to the problem of, given a hypergraph $H = (V, E, W)$, decide if for all non-empty $X \subseteq V$ the distance function δ_X is well-defined, i.e., for all $y \in V$ there exists a shortest hyperpath from X to y. To that end, we need the following lemmas:

Lemma 8. *Let $H = (V, E, W)$ be a hypergraph and $X \subseteq V$, $X \neq \emptyset$ be a set of vertices such that $-\infty < \delta_X(y) < +\infty$ for all $y \in V$. If $(Z, y) \in E$, then $\delta_X(y) \leq W(Z, y) + \max_{z \in Z}(\delta_X(z))$.*

Proof. By hypothesis for all $y \in V$ we have $-\infty < \delta_X(y) < +\infty$. Thus, in particular, for all $z \in Z$ there exists a hyperpath t_z from X to z such that $\omega(t_z) = \delta_X(z)$. Now the tree t with root (Z, y) and children the trees t_z for each $z \in Z$ is a hyperpath from X to y. So $\delta_X(y) \leq \omega(t) = W(Z, y) + \max_{z \in Z}(\omega(t_z)) = W(Z, y) + \max_{z \in Z}(\delta_X(z))$. □

Lemma 9. *Let $H = (V, E, W)$ be a hypergraph and $\alpha : V \to \mathbb{Z}$ be such that $\alpha(y) \leq \max_{z \in Z}(\alpha(z)) + W(Z, y)$ for all hyperedges $(Z, y) \in E$. If t is a hyperpath from a non-empty $X \subseteq V$ to $y \in V$, then $\alpha(y) \leq \max_{x \in X}(\alpha(x)) + \omega(t)$.*

Proof. Let us prove it by induction over the depth of t. In the base case $t = \emptyset$, and therefore $y \in X$. Since $\omega(\emptyset) = 0$, trivially $\alpha(y) \leq \max_{x \in X}(\alpha(x)) = \max_{x \in X}(\alpha(x)) + \omega(\emptyset)$. Now, if t has positive depth, its root is a hyperedge $(Z, y) \in E$, and its children are trees t_1, ..., t_m connecting X to z_1, ..., z_m respectively, where $Z = \{z_1, ..., z_m\}$. By induction hypothesis, for each i in $1 \ldots m$ we have $\alpha(z_i) \leq \max_{x \in X}(\alpha(x)) + \omega(t_i)$. Now:

$$\alpha(y) \leq \max_{1 \leq i \leq n}(\alpha(z_i)) + W(Z, y) \leq \max_{1 \leq i \leq n}(\max_{x \in X}(\alpha(x)) + \omega(t_i)) + W(Z, y) =$$

$$= \max_{x \in X}(\alpha(x)) + \max_{1 \leq i \leq n}(\omega(t_i)) + W(Z, y) = \max_{x \in X}(\alpha(x)) + \omega(t).$$ □

Finally we are in condition to prove the equivalence of the two problems. For convenience, in what remains of this section we assume max-atoms to be of the form $\max_{1 \leq i \leq n}(x_i) + k \geq z$.

Theorem 7. *The Max-atom problem and the problem of well-definedness of the distance functions of all subsets of vertices of a hypergraph are polynomially reducible to each other.*

Proof. First we prove that, given a set S of max-atoms, one can compute in polynomial time a hypergraph $H(S)$ whose distance functions are well-defined if and only if S is satisfiable.

Let S be a set of max-atoms over the variables V. We can assume w.l.o.g. that there exists a variable $x \in V$ such that there are max-atoms $x \geq y \in S$ for every $y \in V$ (adding a fresh variable with these properties preserves satisfiability). The hypergraph $H(S)$ is defined as follows: its set of vertices is V; and for each max-atom $\max_{z \in Z}(z) + k \geq y$, we define a hyperedge $e = (Z, y)$ with weight $W(e) = k$.

Let us see that the distance function δ_x in $H(S)$ is well-defined if and only if S is satisfiable (we write δ_x instead of $\delta_{\{x\}}$ for the sake of clarity). Let us prove that if δ_x is well-defined then S is satisfiable. By construction, for each max-atom $\max_{z \in Z}(z) + k \geq y \in S$ there exists a hyperedge $e = (Z, y)$ in $H(S)$ with weight $W(e) = k$. Now, since δ_x is well-defined and all vertices are hyperconnected to $\{x\}$, by Lemma 8 we have $\max_{z \in Z}(\delta_x(z)) + W(Z, y) \geq \delta_x(y)$, and so $\delta_x \models S$. Let us prove the converse, i.e., that if S is satisfiable then δ_x is well-defined, by contradiction. Let us assume that δ_x is not well-defined and let α be a model of S. Then there is $y \in V$ such that $\delta_x(y) = -\infty$. This implies that for all $w \in \mathbb{R}$ there exists a hyperpath t_w from $\{x\}$ to y such that $\omega(t_w) < w$; in particular, this holds for $w = \alpha(y) - \alpha(x)$. As $\alpha \models S$, by Lemma 9 we have $\alpha(x) + \omega(t_w) \geq \alpha(y)$, i.e., $\omega(t_w) \geq \alpha(y) - \alpha(x)$, which is a contradiction.

Finally, as in $H(S)$ all vertices are hyperconnected to $\{x\}$ by a hyperedge, it is clear that δ_x is well-defined if and only if so is δ_X for all $X \subseteq V$, $X \neq \emptyset$.

Secondly, let us prove that given a hypergraph H, one can compute in polynomial time a set $S(H)$ of max-atoms such that H has a well-defined distance function δ_X for all $X \subseteq V$, $X \neq \emptyset$ if and only if $S(H)$ is satisfiable. Given $H = (V, E, W)$, the variables of $S(H)$ are V, the vertices of H; and for each hyperedge $(Z, y) \in E$, we consider the max-atom $\max_{z \in Z}(z) + W(Z, y) \geq y$. The proof concludes by observing that H has a well-defined distance function δ_X for all $X \subseteq V$, $X \neq \emptyset$ if and only if the same property holds for $H(S(H))$, if and only if $S(H)$ is satisfiable. □

Example 5. The hypergraph corresponding to the set of max-atoms considered in Example 1 is the one shown in Example 4.

6 Conclusions and Future Directions

The contributions of this paper can be summarized as follows:

- First, we have shown that the Max-atom problem is in NP ∩ co-NP. As no PTIME algorithm for solving this problem has been found yet, this is relevant since there are few interesting problems in NP ∩ co-NP that are not known to be polynomial.

- We have given a *weakly* polynomial decision procedure for the problem (when the offsets are integers). This algorithm becomes polynomial under more restrictive conditions on the input, e.g. by imposing a bound on offsets.
- Finally, we have shown the equivalence of deciding the Max-atom problem with two other at first sight unrelated problems: namely, (i) the satisfiability of two-sided linear max-plus systems of equations, used in Control Theory for modeling Discrete Event Systems; and (ii) the existence for a given hypergraph of shortest paths from any non-empty subset of vertices to any vertex. Finding a PTIME algorithm for these problems has been open in the respective areas for more than 30 years, and is still unsolved.

As regards future work, in the short term we would like to find a weakly polynomial algorithm when the offsets may be arbitrary rational numbers. This would perhaps give new insights about the long-term goal of finding a polynomial algorithm for deciding the satisfiability of sets of max-atoms.

As noticed by an anonymous referee, the Max-atom problem is a special case of the problem of finding a super-fixed point of a min-max function. A super-fixed point of a function f on a (partially) ordered set A is an $a \in A$ such that $f(a) \geq a$. Now, for instance, the satisfiability of S in Example 1 from Section 3 is equivalent to finding a super-fixed point of

$$f(u, x, y, z) = (u, u - 10, z, \min(\max(x, y) - 1, \max(x, u) + 25))$$

with respect to the coordinate-wise partial order. More information on min-max functions can be found in [G94]. The referee further mentioned a connection with game theory in [C92]. We gratefully acknowledge these suggestions for future research.

References

[AIN92] Ausiello, G., Italiano, G., Nanni, U.: Optimal traversal of directed hypergraphs. Tech. Rep. TR-92-073, ICSI, Berkeley, CA (1992)

[AKS04] Agrawal, M., Kayal, N., Saxena, N.: PRIMES is in P. Annals of Mathematics 160(2), 781–793 (2004)

[BNRC08] Bezem, M., Nieuwenhuis, R., Rodriguez-Carbonell, E.: Exponential behaviour of the Butkovič-Zimmermann algorithm for solving two-sided linear systems in max-algebra. Discrete Applied Mathematics (to appear)

[BZ06] Butkovič, P., Zimmermann, K.: A strongly polynomial algorithm for solving two-sided linear systems in max-algebra. Discrete Applied Mathematics 154(3), 437–446 (2006)

[C92] Condon, A.: The complexity of stochastic games. Information and Computation 96(2), 203–224 (1992)

[GLPN93] Gallo, G., Longo, G., Pallottino, S., Nguyen, S.: Directed hypergraphs and applications. Discrete Applied Mathematics 42, 177–201 (1993)

[G94] Gunawardena, J.: Min-max functions. Discrete Events Dynamic Systems 4, 377–406 (1994)

[NOT06] Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT Modulo Theories: from an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). Journal of the ACM 53(6), 937–977 (2006)

Towards Practical Feasibility of Core Computation in Data Exchange*

Reinhard Pichler and Vadim Savenkov

Vienna University of Technology

Abstract. Core computation in data exchange is concerned with materializing the minimal target database for a given source database. Gottlob and Nash have recently shown that the core can be computed in polynomial time under very general conditions. Nevertheless, core computation has not yet been incorporated into existing data exchange tools. The principal aim of this paper is to make a big step forward towards the practical feasibility of core computation in data exchange by developing an improved algorithm and by presenting a prototype implementation of our new algorithm.

1 Introduction

Data exchange is concerned with the transfer of data between databases with different schemas. This transfer should be performed so that the source-to-target dependencies (STDs) establishing a mapping between the two schemas are satisfied. Moreover, the target database may also impose additional integrity constraints, called target dependencies (TDs). As STDs and TDs, we consider so-called *embedded dependencies* [1], which are first-order formulae of the form $\forall \boldsymbol{x}\,(\phi(\boldsymbol{x}) \rightarrow \exists \boldsymbol{y}\ \psi(\boldsymbol{x}, \boldsymbol{y}))$ where ϕ and ψ are conjunctions of atomic formulas or equalities, and all variables in $\boldsymbol{x}$ do occur in $\phi(\boldsymbol{x})$. Throughout this paper, we shall omit the universal quantifiers. By convention, all variables occurring in the premise are universally quantified. Moreover, we shall often also omit the existential quantifiers. By convention, all variables occurring in the conclusion only are existentially quantified over the conclusion. We shall thus use the notations $\phi(\boldsymbol{x}) \rightarrow \psi(\boldsymbol{x}, \boldsymbol{y})$ and $\phi(\boldsymbol{x}) \rightarrow \exists \boldsymbol{y}\ \psi(\boldsymbol{x}, \boldsymbol{y})$ interchangeably for the above formula.

The source schema $\mathbf{S}$ and the target schema $\mathbf{T}$ together with the set Σ_{st} of STDs and the set Σ_t of TDs constitute the *data exchange setting* $(\mathbf{S}, \mathbf{T}, \Sigma_{st}, \Sigma_t)$. Following [2,3], we consider dependencies of the following forms: Each STD is a *tuple generating dependency* (TGD) [4] of the form $\phi_{\mathbf{S}}(\boldsymbol{x}) \rightarrow \psi_{\mathbf{T}}(\boldsymbol{x}, \boldsymbol{y})$, where $\phi_{\mathbf{S}}(\boldsymbol{x})$ is a conjunction of atomic formulas over $\mathbf{S}$ and $\psi_{\mathbf{T}}(\boldsymbol{x}, \boldsymbol{y})$ is a conjunction of atomic formulas over $\mathbf{T}$. Each TD is either a TGD, of the form $\phi_{\mathbf{T}}(\boldsymbol{x}) \rightarrow \psi_{\mathbf{T}}(\boldsymbol{x}, \boldsymbol{y})$ or an *equality generating dependency* (EGD) [4] of the form $\phi_{\mathbf{T}}(\boldsymbol{x}) \rightarrow (x_i = x_j)$. In these dependencies, $\phi_{\mathbf{T}}(\boldsymbol{x})$ and $\psi_{\mathbf{T}}(\boldsymbol{x}, \boldsymbol{y})$ are conjunctions of atomic formulas over $\mathbf{T}$, and x_i, x_j are among the variables in $\boldsymbol{x}$. An important special case of

* This work was supported by the Austrian Science Fund (FWF), project P20704-N18.

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 62–78, 2008.

TGDs are *full TGDs*, which have no (existentially quantified) variables $\boldsymbol{y}$, i.e. we have $\phi_{\mathbf{S}}(\boldsymbol{x}) \rightarrow \psi_{\mathbf{T}}(\boldsymbol{x})$ and $\phi_{\mathbf{T}}(\boldsymbol{x}) \rightarrow \psi_{\mathbf{T}}(\boldsymbol{x})$, respectively.

The *data exchange problem* for a data exchange setting $(\mathbf{S}, \mathbf{T}, \Sigma_{st}, \Sigma_t)$ is the task of constructing a target instance J for a given source instance I, s.t. all STDs Σ_{st} and TDs Σ_t are satisfied. Such a J is called a *solution*. Typically, the number of possible solutions to a data exchange problem is infinite.

Example 1. Suppose that the source instance consists of two relations Tutorial(course, tutor): {('java', 'Yves')} and BasicUnit(course): {'java'}. Moreover, let the target schema have four relation symbols NeedsLab(id_tutor,lab), Tutor(idt,tutor), Teaches(id_tutor, id_course) and Course(idc,course). Now suppose that we have the following STDs:

1. $\mathsf{BasicUnit}(C) \rightarrow \mathsf{Course}(Idc, C)$.
2. $\mathsf{Tutorial}(C, T) \rightarrow \mathsf{Course}(Idc, C), \mathsf{Tutor}(Idt, T), \mathsf{Teaches}(Idt, Itc)$.

and suppose that the TDs are given by the two TGDs:

3. $\mathsf{Course}(Idc, C) \rightarrow \mathsf{Tutor}(Idt, T), \mathsf{Teaches}(Idt, Idc)$.
4. $\mathsf{Teaches}(Idt, Idc) \rightarrow \mathsf{NeedsLab}(Idt, L)$.

Then the following instances are all valid solutions:

$J = \{\mathsf{Course}(C_1, \text{'java'}), \mathsf{Tutor}(T_2,N), \mathsf{Teaches}(T_2,C_1), \mathsf{NeedsLab}(T_2,L_2),$
$\mathsf{Course}(C_2, \text{'java'}), \mathsf{Tutor}(T_1,\text{'Yves'}), \mathsf{Teaches}(T_1,C_2), \mathsf{NeedsLab}(T_1,L_1)\},$

$J_c = \{\mathsf{Course}(C_1,\text{'java'}), \mathsf{Tutor}(T_1,\text{'Yves'}), \mathsf{Teaches}(T_1,C_1), \mathsf{NeedsLab}(T_1,L_1)\},$

$J' = \{\mathsf{Course}(\text{'java'},\text{'java'}), \mathsf{Tutor}(T_1,\text{'Yves'}), \mathsf{Teaches}(T_1,\text{'java'}), \mathsf{NeedsLab}(T_1,L_1)\}$

A natural requirement (proposed in [2]) on the solutions is *universality*, that is, there should be a homomorphism from the materialized solution to any other possible solution. Note that J' in Example 1 is not universal, since there exists no homomorphism $h\colon J' \rightarrow J$. Indeed, a homomorphism maps any constant onto itself; thus, the fact $\mathsf{Course}(\text{'java'},\text{'java'})$ cannot be mapped onto a fact in J.

In general, a data exchange problem has several universal solutions, which may significantly differ in size. However, there is – up to isomorphism – one particular, universal solution, called the *core* [3], which is the most compact one. For instance, solution J_c in Example 1 is a core.

Fagin et al. [3] gave convincing arguments that the core should be the database to be materialized. In general, computing the core of a graph or a structure is NP-complete [5]. However, Gottlob and Nash [6] showed that the core of the target database in data exchange can be computed in polynomial time under very general conditions. Despite this favorable complexity result, core computation has not yet been incorporated into existing data exchange tools. This is mainly due to the following reasons: (1) Despite the theoretical tractability of core computation, we are still far away from a practically efficient implementation of core computation. In fact, no implementation at all has been reported so far. (2) The core computation looks like a separate technology which cannot be easily integrated into existing database technology.

Results. The main contribution of this work is twofold:

(1) We present an enhanced version of the FindCore algorithm, which we shall refer to as FindCore^E. One of the specifics of FindCore is that EGDs in the target dependencies are simulated by TGDs. As a consequence, the core computation becomes an integral part of finding any solution to the data exchange problem. The most significant advantage of our FindCore^E algorithm is that it avoids the simulation of EGDs by TGDs. The activities of solving the data exchange problem and of computing the core are thus fully uncoupled. The core computation can then be considered as an optional add-on feature of data exchange which may be omitted or deferred to a later time (e.g., to periods of low database user activity). Moreover, the direct treatment of EGDs leads to a performance improvement of an order of magnitude. Another order of magnitude can be gained by approximating the core. Our experimental results suggest that the partial execution of the core computation may already yield a very good approximation to the core. Since all intermediate instances computed by our FindCore^E algorithm are universal solutions, one may stop the core computation at any time and content oneself with an approximation to the core.

(2) We also report on a proof-of-concept implementation of our enhanced algorithm. It is built on top of a relational database system and mimics data exchange-specific features by automatically generated views and SQL queries. This shows that the integration of core computation into existing database technology is clearly feasible. The lessons learned from the experiments with this implementation yield important hints concerning future improvements.

Due to lack of space, most proofs are sketched or even omitted in this paper. For full proofs, we refer to [7].

2 Preliminaries

2.1 Basic Notions

Schemas and instances. A *schema* $\sigma = \{R_1, \ldots, R_n\}$ is a set of relation symbols R_i with fixed arities. An *instance* over a schema σ consists of a relation for each relation symbol in σ, s.t. both have the same arity. Tuples of the relations may contain two types of *terms*: *constants* and *variables*. The latter are also called *labeled nulls*. Two labeled nulls are equal iff they have the same label. For every instance J, we write $\mathrm{dom}(J)$, $\mathrm{var}(J)$, and $\mathrm{const}(J)$ to denote the set of terms, variables, and constants, respectively, of J. Clearly, $\mathrm{dom}(J) = \mathrm{var}(J) \cup \mathrm{const}(J)$ and $\mathrm{var}(J) \cap \mathrm{const}(J) = \emptyset$. If a tuple $(x_1, x_2, \ldots, x_n)$ belongs to the relation R, we say that J contains the *fact* $R(x_1, x_2, \ldots, x_n)$. We write $\boldsymbol{x}$ for a tuple $(x_1, x_2, \ldots, x_n)$ and if $x_i \in X$, for every i, then we also write $\boldsymbol{x} \in X$ instead of $\boldsymbol{x} \in X^n$. Likewise, we write $r \in \boldsymbol{x}$ if $r = x_i$ for some i. Let Σ be an arbitrary set of dependencies and J an instance. We write $J \models \Sigma$ to denote that the instance J satisfies Σ. In a data exchange setting $(\mathbf{S}, \mathbf{T}, \Sigma_{st}, \Sigma_t)$, the source schema $\mathbf{S}$ and the target schema $\mathbf{T}$ have no relation symbols in common. In a source instance I, no variables are allowed, i.e., $\mathrm{dom}(I) = \mathrm{const}(I)$.

Chase. The data exchange problem can be solved by the *chase* [4], which iteratively introduces new facts or equates terms until all desired dependencies are fulfilled. More precisely, let Σ contain a TGD $\tau\colon \phi(\boldsymbol{x}) \rightarrow \psi(\boldsymbol{x},\boldsymbol{y})$, s.t. $I \models \phi(\boldsymbol{a})$ for some assignment $\boldsymbol{a}$ on $\boldsymbol{x}$ and $I \not\models \exists \boldsymbol{y}\psi(\boldsymbol{a},\boldsymbol{y})$. Then we have to extend I with facts corresponding to $\psi(\boldsymbol{a},\boldsymbol{z})$, where the elements of $\boldsymbol{z}$ are fresh labeled nulls. Likewise, suppose that Σ contains an EGD $\tau\colon \phi(\boldsymbol{x}) \rightarrow x_i = x_j$, s.t. $I \models \phi(\boldsymbol{a})$ for some assignment $\boldsymbol{a}$ on $\boldsymbol{x}$. This EGD enforces the equality $a_i = a_j$. We thus choose a variable v among a_i, a_j and replace *every occurrence* of v in I by the other term; if $a_i, a_j \in \mathrm{const}(I)$ and $a_i \neq a_j$, the chase halts with *failure*. The result of chasing I with dependencies Σ is denoted as I^{Σ}.

A sufficient condition for the termination of the chase is that the TGDs be *weakly acyclic* (see [8,2]). This property is formalized as follows. For a dependency set Σ, construct a *dependency graph* G^D whose vertices are *fields* R^i where i denotes a position (an "attribute") of relation R. Let $\phi(\boldsymbol{x}) \rightarrow \psi(\boldsymbol{x},\boldsymbol{y})$ be a TGD in Σ and suppose that some variable $x \in \boldsymbol{x}$ occurs in the field R^i. Then the edge $\left(R^i, S^j\right)$ is present in G^D if either (1) x also occurs in the field S^j in $\psi(\boldsymbol{x},\boldsymbol{y})$ or (2) x occurs in some other field T^k in $\psi(\boldsymbol{x},\boldsymbol{y})$ and there is a variable $y \in \boldsymbol{y}$ in the field S^j in $\psi(\boldsymbol{x},\boldsymbol{y})$. Edges resulting from rule (2) are called *special*.

A set of TGDs is *weakly acyclic* if there is no cycle containing a special edge. Obviously, the set of STDs is always weakly acyclic, since the dependency graph contains only edges from fields in the source schema to fields in the target schema. We thus consider data exchange settings $(\mathbf{S}, \mathbf{T}, \Sigma_{st}, \Sigma_t)$ where Σ_{st} is a set of TGDs and Σ_t is a set of EGDs and *weakly acyclic* TGDs.

Figure 1 shows the dependency graph for the *target* TGDs in Example 1. Special edges are marked with *. Source-to-target TGDs are omitted, since they can never produce a cycle. Figure 1 contains two kinds of vertices: the ovals, labelled by attributes of target relations, are the actual vertices of the dependency graph. The rectangles, labelled by relation names, were inserted to improve the readability. Rather than adding the relation names to the attributes in the labels of the oval vertices, we have connected each attribute to the rectangular vertex with the corresponding relation name. Clearly, this dependency graph has no cycle containing a special edge. Hence, these TGDs are weakly acyclic.

Universal solutions and core. Let I,I' be instances. A *homomorphism* $h\colon I \rightarrow I'$ is a mapping $\mathrm{dom}(I) \rightarrow \mathrm{dom}(I')$, s.t. (1) whenever $R(\boldsymbol{x}) \in I$, then $R(h(\boldsymbol{x})) \in I'$, and (2) for every constant c, $h(c) = c$. An *endomorphism* is a homomorphism $I \rightarrow I$, and a *retraction* r is an idempotent endomorphism, i.e. $r \circ r = r$. An endomorphism or a retraction is *proper* if it is not surjective (for finite instances, this is equivalent to being not injective). The image $r(I)$ under a retraction r is called a *retract* of I. An instance is called a *core* if it has no proper retractions. A core C of an instance I is a retract of I, s.t. C is a core. Cores of an instance I are unique up to isomorphism. We can therefore speak about *the core* of I.

Consider a data exchange setting where Σ_{st} is a set of TGDs and Σ_t is a set of EGDs and weakly acyclic TGDs. Then the solution to a source instance S can be computed as follows: We start off with the instance $(S, \emptyset)$, i.e., the source instance is S and the target instance is initially empty. Chasing $(S, \emptyset)$

with Σ_{st} yields the instance (S, T), where T is called the *preuniversal instance*. This chase always succeeds since Σ_{st} contains no EGDs. Then T is chased with Σ_t. This chase may fail because of the EGDs in Σ_t. If the chase succeeds, then we end up with $U = T^{\Sigma_t}$, which is referred to as the *canonical universal solution*. Both T and U can be computed in polynomial time w.r.t. the size of the source instance [2].

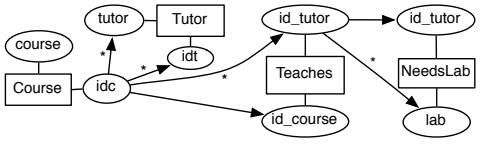


Fig. 1. Dependency graph

Depth, height, width, blocks. Let Σ be a set of dependencies with dependency graph G^D. The *depth* of a field R^j of a relation symbol R is the maximal number of special edges in any path of G^D that ends in R^j. The depth of Σ is the maximal depth of any field in Σ. Given a dependency τ: $\phi(\boldsymbol{x}) \to \psi(\boldsymbol{x}, \boldsymbol{y})$ in Σ, we define the *width* of τ to be $|\boldsymbol{x}|$, and the *height* as $|\boldsymbol{y}|$. The width (resp. the height) of Σ is the maximal width (resp. height) of the dependencies in Σ.

Core computation is essentially a search for appropriate homomorphisms, whose key complexity factor is the *block size* [3]. It is defined as follows: The *Gaifman graph* $\mathcal{G}(I)$ of an instance I is an undirected graph whose vertices are the variables of I and, whenever two variables v_1 and v_2 share a tuple in I, there is an edge (v_1, v_2) in $\mathcal{G}(I)$. A *block* is a connected component of $\mathcal{G}(I)$. Every variable v of I belongs exactly to one block, denoted as $\text{block}(v, I)$. The *block size* of instance I is the maximal number of variables in any of its blocks.

Theorem 1. [3] *Let A and B be instances, and suppose that* $\text{blocksize}(A) \leq c$ *holds. Then the check if a homomorphism $h\colon A \to B$ exists and, if so, the computation of h can both be done in time $O(|A| \cdot |B|^c)$.*

Theorem 2. [3] *If Σ_{st} is a set of STDs of height e, S is ground, and $(S, T) = (S, \emptyset)^{\Sigma_{st}}$, then* $\text{blocksize}(T) \leq e$.

Sibling, parent, ancestor. Consider the chase of the preuniversal instance T with TDs Σ_t and suppose that $\boldsymbol{y}$ is a tuple of variables created by enforcing a TGD $\phi(\boldsymbol{x}) \to \psi(\boldsymbol{x}, \boldsymbol{y})$ in Σ_t, s.t. the precondition $\phi(\boldsymbol{x})$ was satisfied with a tuple $\boldsymbol{a}$. Then the elements of $\boldsymbol{y}$ are *siblings* of each other; every variable of $\boldsymbol{a}$ is a *parent* of every element of $\boldsymbol{y}$; and the *ancestor* relation is the transitive closure of the parent relation.

2.2 Core Computation with FindCore

In this section, we recall the FindCore algorithm of [6]. To this end, we briefly explain the main ideas underlying the steps (1) – (11) of this algorithm.

The chase. FindCore starts in (1) with the computation of the preuniversal instance. But then, rather than directly computing the canonical universal solution by chasing T with Σ_t, the EGDs in Σ_t are simulated by TGDs. Hence,

in (2), the set Σ_t of EGDs and TGDs over the signature τ is transformed into the set $\bar{\Sigma}_t$ of TGDs over the signature $\tau \cup E$, where E (encoding equality) is a binary relation not present in τ. The transformation proceeds as follows:

1. Replace all equations $x = y$ with $E(x, y)$, turning every EGD into a TGD.
2. Add *equality* constraints (symmetry, transitivity, reflexivity): (i) $E(x, y) \rightarrow E(y, x)$; (ii) $E(x, y), E(y, z) \rightarrow E(x, z)$; and (iii) $R(x_1, \ldots, x_k) \rightarrow E(x_i, x_i)$ for every $R \in \tau$ and $i \in \{1, 2, \ldots, k\}$ where k is the arity of R.
3. Add *consistency* constraints: $R(x_1, \ldots, x_k), E(x_i, y) \rightarrow R(x_1, \ldots, y, \ldots, x_k)$ for every $R \in \tau$ and $i \in \{1, 2, \ldots, k\}$.

Procedure FindCore
Input: Source ground instance S
Output: Core of a universal solution for S

```
(1)  Chase (S,∅) with Σ_st
     to obtain (S,T) := (S,∅)^{Σ_st};
(2)  Compute Σ̄_t from Σ_t;
(3)  Chase T with Σ̄_t (using a nice order)
     to get U := T^{Σ̄_t};
(4)  for each x ∈ var(U), y ∈ dom(U), x ≠ y do
(5)      Compute T_xy;
(6)      Look for h: T_xy → U s.t. h(x) = h(y);
(7)      if there is such h then
(8)          Extend h to an endomorphism h' on U;
(9)          Transform h' into a retraction r;
(10)         Set U := r(U);
(11) return U.
```

Even if Σ_t was weakly acyclic, $\bar{\Sigma}_t$ may possibly not be so. Hence, a special *nice chase order* is defined in [6] which ensures termination of the chase by $\bar{\Sigma}_t$. It should be noted that U computed in (3) is not a universal solution since, in general, the EGDs of Σ_t are not satisfied. Their enforcement happens as part of the core computation.

Retractions. The FindCore algorithm computes the core by computing nested retracts. This is motivated by the following properties of retractions: (1) embedded dependencies are closed under retractions and (2) any proper endomorphism can be efficiently transformed into a retraction [6]:

Theorem 3. [6] *Let $r\colon A \rightarrow A$ be a retraction with $B = r(A)$ and let Σ be a set of embedded dependencies. If $A \models \Sigma$, then $B \models \Sigma$.*

Theorem 4. [6] *Given an endomorphism $h\colon A \rightarrow A$ such that $h(x) = h(y)$ for some $x, y \in \mathrm{dom}(A)$, there is a proper retraction r on A s.t. $r(x) = r(y)$. Such a retraction can be found in time $O(|\mathrm{dom}(A)|^2)$.*

Note that U after step (3) clearly satisfies the dependencies Σ_{st} and $\bar{\Sigma}_t$. Steps (4) – (8), which will be explained below, search for a proper endomorphism h on U. If this search is successful, we use Theorem 4 to turn h into a retraction r in step (9) and replace U by $r(U)$ in step (10). By Theorem 3 we know that Σ_{st} and $\bar{\Sigma}_t$ are still satisfied.

Searching for proper endomorphisms. At every step of the descent to the core, the FindCore algorithm attempts to find a proper endomorphism for the current instance U in the steps (5) – (8) of the algorithm. Given a variable x and another domain element y, we try to find an endomorphism which equates x and y. However, by Theorem 1, the time needed to find an appropriate homomorphism may be exponential w.r.t. the block size. The key idea in FindCore

is, therefore, to split the search for a proper endomorphism into two steps: For given x and y, there exists an instance T_{xy} (defined below) whose block size is bounded by a constant depending only on $\Sigma_{st} \cup \Sigma_t$. So we first search for a homomorphism $h\colon T_{xy} \to U$ with $h(x) = h(y)$; and then h is extended to a homomorphism $h\colon U \to U$, s.t. $h(x) = h(y)$ still holds. Hence, h is still non-injective and, thus, h is a *proper* endomorphism, since we only consider finite instances. The properties of T_{xy} and the existence of an extension h' of h are governed by the following results from [6]:

Lemma 1. [6] *For every weakly acyclic set Σ of TGDs, instance T and $x, y \in dom(T^\Sigma)$, there exist constants b, c which depend only on Σ and an instance T_{xy} satisfying* (1) $x, y \in \mathrm{dom}(T_{xy})$, (2) $T \subseteq T_{xy} \subseteq T^\Sigma$, (3) $\mathrm{dom}(T_{xy})$ *is closed under parents and siblings, and* (4) $|\mathrm{dom}(T_{xy})| \leq |\mathrm{dom}(T)| + b$. *Moreover, T_{xy} can be computed in time $O(|\mathrm{dom}(T)|^c)$.*

Theorem 5. (Lifting) [6] *Let T^Σ be a universal solution of a data exchange problem obtained by chasing a preuniversal instance T with the weakly acyclic set Σ of target TGDs. If B and W are instances such that:* (1) $B \models \Sigma$, (2) $T \subseteq W \subseteq T^\Sigma$, *and* (3) $\mathrm{dom}(W)$ *is closed under ancestors and siblings, then any homomorphism $h\colon W \to B$ can be extended in time $O(|\mathrm{dom}(T)|^b)$ to a homomorphism $h'\colon T^\Sigma \to B$ where b depends only on Σ.*

Summary. Recall that the predicate E simulates equality. Hence, if step (3) of the algorithm generates a fact $E(a_i, a_j)$ with $a_i \neq a_j$ then the data exchange problem has no solution. Otherwise, the loop in steps (4) – (10) tries to successively shrink $\mathrm{dom}(U)$. When no further shrinking is possible, then the core is reached. In fact, it is proved in [6] that such a minimal instance U resulting from FindCore indeed satisfies all the EGDs. Hence, U minus all auxiliary facts with leading symbol E constitutes the core of a universal solution. Moreover, it is proved in [6] that the entire computation fits into $O(|\mathrm{dom}(S)|^b)$ time for some constant b which depends only on the dependencies $\Sigma_{st} \cup \Sigma_t$.

3 Enhanced Core Computation

The crucial point of our enhanced algorithm FindCore^E is the *direct* treatment of the EGDs, rather than simulating them by TGDs. Hence, our algorithm produces the canonical universal solution U first (or detects that no solution exists), and then successively minimizes U to the core. On the surface, our FindCore^E algorithm proceeds exactly as the FindCore algorithm from Section 2.2 algorithm, i.e.: `(i)` compute an instance T_{xy}; `(ii)` search for a non-injective homomorphism $h\colon T_{xy} \to U$; `(iii)` lift h to a proper endomorphism $h'\colon U \to U$; and `(iv)` construct a proper retraction r from h'.

Actually, the construction of a retraction r via Theorem 4 and the closure of embedded dependencies w.r.t. retractions according to Theorem 3 are not affected by the application of the EGDs. In contrast, the first 3 steps above require significant adaptations in order to cope with EGDs:

(i) T_{xy} in Section 2.2 is obtained by considering only a small portion of the target chase, thus producing a subinstance of U. Now that EGDs are involved, the domain of U may no longer contain all elements that were present in T or in some intermediate result of the chase. Hence, we need to define T_{xy} differently.

(ii) The computational cost of the search for a homomorphism $h\colon T_{xy} \to U$ depends on the block size of T_{xy} which in turn depends on the block size of the preuniversal instance T. EGDs have a positive effect in that they eliminate variables, thus reducing the size of a single block. However, EGDs may also merge different blocks of T. Hence, without further measures, this would destroy the tractability of the search for a homomorphism $h\colon T_{xy} \to U$.

(iii) Since T_{xy} is defined differently from Section 2.2, also the lifting of $h\colon T_{xy} \to U$ to a proper endomorphism $h'\colon U \to U$ has to be modified. Moreover, it will turn out that a completely new approach is needed to prove the correctness of this lifting. The details of the FindCore^E algorithm are worked out below.

Introduction of an id. Chasing with EGDs results in the substitution of variables. Hence, the application of an EGD to an instance J produces a syntactically different instance J'. However, we find it convenient to regard the instance J' after enforcement of an EGD as a *new version* of the instance J rather than as a completely new instance. In other words, the substitution of a variable produces new versions of facts that have held that variable, but the facts themselves persist. We formalize this idea as follows: Given a data exchange setting $S = (\mathbf{S}, \mathbf{T}, \Sigma_{st}, \Sigma_t)$, we define an *id-aware data exchange setting* S^{id} by augmenting each relation $R \in \mathbf{T}$ with an additional *id* field inserted at position 0. Hence, in the atoms of the conclusions of STDs and in all atoms occurring in TDs, we have to add a unique existentially-quantified variable at position 0. For example, the source-to-target TGD $\tau\colon S(x) \to R(x, y)$ is transformed into $\tau^{id}\colon S(x) \to R^{id}(t, x, y)$ for some fresh variable t.

These changes neither have an effect on the chase nor on the core computation, as no rules rely on values in the added columns. It is immediate that a fact $R(x_1, x_2, \ldots, x_n)$ is present in the target instance at some phase of solving the original data exchange problem iff the fact $R^{id}(id, x_1, x_2, \ldots, x_n)$ is present at the same phase of solving its *id-aware* version. In fact, this modification does not even need to be implemented – we just introduce it to allow the discussion about facts in an unambiguous way.

During the chase, every fact of the target instance is assigned a unique *id* variable, which is never substituted by an EGD. We can therefore identify a fact with this variable: (1) if $R^{id}(t_1, x_1, \ldots, x_n)$ is a fact of a target instance $\mathbf{T}$, then we refer to it as *fact* t_1; (2) we define equality on facts as equality between their *id* terms: $R^{id}(t_1, x_1, \ldots, x_n) = R^{id}(t_2, y_1, \ldots, y_n)$ iff $t_1 = t_2$.

We also define a *position* by means of the *id* of a fact plus a positive integer indicating the place of this position inside the fact. Thus, if J is an instance and $R(id_R, x_1, x_2, \ldots, x_n)$ is an *id-aware* version of $R(x_1, \ldots, x_n) \in J$, then we say that the term x_i occurs at the position (id_R, i) in J.

Source position and origin. By the above considerations, facts and positions in an *id-aware data exchange setting*, persist in the instance once they have

been created – in spite of possible modifications of the variables. New facts and, therefore, new positions in the target instance are introduced by TGDs. If a position $p = (id_R, i)$ occurring in the fact $R(id_R, x_1, \ldots, x_n)$ was created to hold a fresh variable, we call p *native* to its fact id_R. Otherwise, if an already existing variable was copied from some position p' in the premise of the TGD to p, then we say that p is *foreign* to its fact id_R. Moreover, we call p' the *source position* of p. Note that there may be multiple choices for a source position. For instance, in the case of the TGD $R(y, x) \wedge S(x) \rightarrow P(x)$: a term of $P/1$ may be copied either from $R/2$ or from $S/1$. Any possibility can be taken in such a case: the choice is *don't care non-deterministic*.

Of course, a source position may itself be foreign to its fact. Tracing the chain of source positions back until we reach a native position leads to the notion of *origin position*, which we define recursively as follows: If a position $p = (id_R, i)$ is native to the fact $R(id_R, x_1, \ldots, x_n)$, then its origin position is p itself. Otherwise, if p is foreign, then the origin of p is the origin of a *source position* of p.

The fact holding the origin position of p is referred to as the *origin fact of the position* p. Finally, we define the *origin fact of a variable* x, denoted as $Origin_x$, as the origin fact of one of the positions where it was first introduced (again in a don't care non-deterministic way).

Example 2. Let $J = \{S(id_{S1}, x_1, y_1)\}$ be a preuniversal instance, and consider the TDs $\{S(id_S, x, y) \rightarrow P(id_P, y, z); P(id_P, y, z) \rightarrow Q(id_Q, y, v)\}$ yielding the canonical solution $J^\Sigma = \{S(id_{S1}, x_1, y_1), P(id_{P1}, y_1, z_1), Q(id_{Q1}, y_1, v_1)\}$ in Figure 2. Every position of J is native, being created by the source-to-target chase, which never copies labeled nulls. Thus the origin positions of $(id_{S1}, 1)$ and $(id_{S1}, 2)$ are these positions themselves. The latter is also the origin position for the two foreign positions $(id_{P1}, 1)$ and $(id_{Q1}, 1)$, introduced by the target chase (foreign positions are dashed in the figure). The remaining two positions of the facts id_{P1} and id_{Q1} are native. The origin positions of the variables are: $(id_{S1}, 1)$ for x_1, $(id_{S1}, 2)$ for y_1, $(id_{P1}, 2)$ for z_1, and $(id_{Q1}, 2)$ for v_1.

Lemma 2. *Let I be an instance. Moreover, let p be a position in I and o_p its origin position. Then p and o_p always contain the same term.*

Normalization of TGDs. Let $\tau\colon \phi(\boldsymbol{x}) \rightarrow \psi(\boldsymbol{x}, \boldsymbol{y})$ be a non-full TGD, i.e., $\boldsymbol{y}$ is non-empty. Then we can set up the *Gaifman graph* $\mathcal{G}(\tau)$ of the atoms in the conclusion $\psi(\boldsymbol{x}, \boldsymbol{y})$, considering only the new variables $\boldsymbol{y}$, i.e., $\mathcal{G}(\tau)$ contains as vertices the variables in $\boldsymbol{y}$. Moreover, two variables y_i and y_j are adjacent

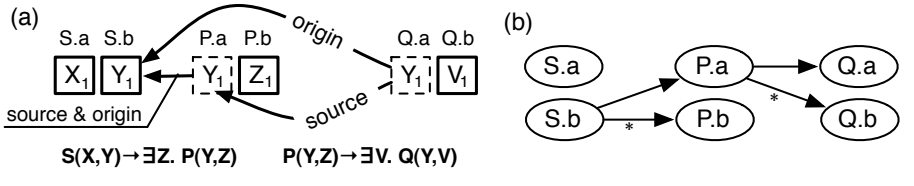


Fig. 2. Positions of the instance J^Σ (a) and the dependency graph of Σ (b)

(by slight abuse of notation, we identify vertices and variables), if they jointly occur in some atom of $\psi(\boldsymbol{x}, \boldsymbol{y})$. Let $\mathcal{G}(\tau)$ contain the connected components $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_n$. Then the conclusion is of the form $\psi(\boldsymbol{x}, \boldsymbol{y}) = \psi_0(\boldsymbol{x}) \wedge \psi_1(\boldsymbol{x}, \boldsymbol{y}_1) \wedge \cdots \wedge \psi_n(\boldsymbol{x}, \boldsymbol{y}_n)$, where the subformula $\psi_0(\boldsymbol{x})$ contains all atoms of $\psi(\boldsymbol{x}, \boldsymbol{y})$ without variables from $\boldsymbol{y}$ and each subformula $\psi_i(\boldsymbol{x}, \boldsymbol{y}_i)$ contains exactly the atoms of $\psi(\boldsymbol{x}, \boldsymbol{y})$ containing at least one variable from the connected component $\boldsymbol{y}_i$.

Now let the full TGD τ_0 be defined as $\tau_0 \colon \phi(\boldsymbol{x}) \rightarrow \psi_0(\boldsymbol{x})$ and let the non-full TGDs τ_i with $i \in \{1, \ldots, n\}$ be defined as $\tau_i \colon \phi(\boldsymbol{x}) \rightarrow \psi_i(\boldsymbol{x}, \boldsymbol{y}_i)$. Then τ is clearly logically equivalent to the conjunction $\tau_0 \wedge \tau_1 \wedge \cdots \wedge \tau_n$. Hence, τ in the set Σ_t of target dependencies may be replaced by $\tau_0, \tau_1, \ldots, \tau_n$.

We say that Σ_t is in *normal form* if every TGD τ in Σ_t is either full or its Gaifman graph $\mathcal{G}(\tau)$ has exactly 1 connected component. By the above considerations, we will henceforth assume w.l.o.g., that Σ_t is in normal form.

Example 3. The non-full TGD $\tau \colon S(x, y) \rightarrow \exists z, v(P(x, z) \wedge R(x, y) \wedge Q(y, v))$ is logically equivalent to the conjunction of the three TGDs: $\tau_0 \colon S(x, y) \rightarrow R(x, y)$, $\tau_1 \colon S(x, y) \rightarrow \exists z\ P(x, z)$, and $\tau_2 \colon S(x, y) \rightarrow \exists v\ Q(y, v)$. Clearly, these dependencies τ_0, τ_1, and τ_2 are normalized in the above sense.

Extension of the parent and sibling relation to facts. Let I be an instance after the j^{th} chase step and suppose that in the next chase step, the *non-full* TGD $\tau \colon \phi(\boldsymbol{x}) \rightarrow \psi(\boldsymbol{x}, \boldsymbol{y})$ is enforced, i.e.: $I \models \phi(\boldsymbol{a})$ for some assignment $\boldsymbol{a}$ on $\boldsymbol{x}$ and $I \not\models \exists \boldsymbol{y} \psi(\boldsymbol{a}, \boldsymbol{y})$, s.t. the facts corresponding to $\psi(\boldsymbol{a}, \boldsymbol{z})$, where the elements of $\boldsymbol{z}$ are fresh labeled nulls, are added. Let t be a fact introduced by this chase step, i.e., t is an atom of $\psi(\boldsymbol{a}, \boldsymbol{z})$. Then all other facts introduced by the same chase step (i.e., all other atoms of $\psi(\boldsymbol{a}, \boldsymbol{z})$) are the *siblings* of t. The *parent* set of a fact t consists of the origin facts for any foreign position in t or in any of its siblings. The *ancestor* relation on facts is the transitive closure of the parent relation. This definition of siblings and parents implies that facts introducing no fresh nulls (since we are assuming the above normal form, these are the facts created by a full TGD) can be neither parents nor siblings.

Recall that we identify facts by their ids rather than by their concrete values. Hence, any substitutions of nulls that happen in the course of the chase do not change the set of siblings, the set of parents, or the set of ancestors of a fact.

Example 4. Let us revisit the two TGDs $S(id_S, x, y) \rightarrow P(id_P, y, z)$ and $P(id_P, y, z) \rightarrow Q(id_Q, y, v)$ from Example 2, see also Figure 2. Although the creation of the atom $Q(y_1, v_1)$ was triggered by the atom $P(y_1, z_1)$, the only parent of $Q(y_1, v_1)$ is the origin fact of y_1, namely $S(x_1, y_1)$.

Some useful notation. To reason about the effects of EGDs, it is convenient to introduce some additional notation, following [3]. Let J be a canonical preuniversal instance and J' the canonical universal solution, resulting from chasing J with a set of target dependencies Σ_t. Moreover, suppose that u is a term which either exists in the domain of J or which is introduced in the course of the chase. Then we write $[u]$ to denote the term to which u is mapped by the chase. More precisely, let $t = S(u_1, u_2, \ldots, u_s)$ be an arbitrary fact, which either exists in J or which is introduced by the chase. Then the same fact t in J' has the form

$S([u_1], [u_2], \ldots, [u_s])$. By Lemma 2, every $[u_i]$ is well-defined, since it corresponds to the term produced by the chase in the corresponding origin position. For any set Σ_t of TDs, constants are mapped onto themselves: $\forall c \in \text{const}(J)\ c = [c]$. For $u, v \in \text{dom}(J)$, we write $u \sim v$ if $[u] = [v]$, i.e. two terms have the same image in J'. If Σ_t contains no EGDs, then $u = [u]$ holds for all $u \in \text{dom}(J)$. Clearly, the mapping $[\cdot]\colon J \to J'$ is a homomorphism.

The following lemma is the basis for constructing a homomorphism $h'\colon T^{\Sigma_{st}} \to U$, analogously to Theorem 5 by extending a homomorphism $h\colon T_{xy} \to U$.

Lemma 3. *For every weakly acyclic set Σ_t of TGDs and EGDs, instance T, and $x, y \in dom(T^{\Sigma_t})$, there exist constants b, c which depend only on $\Sigma = \Sigma_{st} \cup \Sigma_t$ and an instance T_{xy} satisfying* (1) *$Origin_x, Origin_y \subseteq T_{xy}$,* (2) *all facts of T are in T_{xy}, and $T_{xy} \subseteq T^{\Sigma_t}$,* (3) *$T_{xy}$ is closed under parents and siblings over facts, and* (4) *$|\text{dom}(T_{xy})| \leq |\text{dom}(T)| + b$. Moreover, T_{xy} can be computed in time $O(|\text{dom}(T)|^c)$.*

Compared with Lemma 1, we had to redefine the set T_{xy}. Moreover, the unification of variables caused by EGDs in the chase invalidates some essential assumptions in the proof of the corresponding result in [6, Theorem 7]. At any rate, also in our case, the lifting can be performed efficiently:

Theorem 6. *(*Lifting*) Let T^{Σ_t} be a universal solution of a data exchange problem obtained by chasing a preuniversal instance T with the weakly acyclic set Σ_t of TGDs and EGDs. If B and W are instances such that:* (1) *$B \models \Sigma$ with $\Sigma = \Sigma_{st} \cup \Sigma_t$;* (2) *all facts of T are in W (i.e. W contains facts with the same* ids*) and $W \subseteq T^{\Sigma_t}$, and* (3) *W is closed under ancestors and siblings (over facts), then any homomorphism $h\colon W \to B$ can be transformed in time $O(|\text{dom}(T)|^b)$ into a homomorphism $h'\colon T^{\Sigma_t} \to B$, s.t. $\forall x \in dom(h)\colon h(x) = h'(x)$, where b depends only on Σ.*

Proof. Although every fact of T is in W, there may of course be variables in $\text{dom}(T)$ which are not in $\text{dom}(W)$, because of the EGDs. Hence, $\forall x \in \text{dom}(T) \setminus \text{dom}(W)\colon x \neq [x]$, and $\forall x \in \text{dom}(T) \cap \text{dom}(W)\colon x = [x]$.

Suppose that the chase of a preuniversal instance T with Σ_t has length n. Then we write T_s with $0 \leq s \leq n$ to denote the result after step s of the chase. In particular, we have $T_0 = T$ and $T_n = T^{\Sigma_t}$. For every s, we say that a homomorphism $h_s\colon T_s \to B$ is *consistent with* h if $\forall x \in \text{dom}(h_s)$, such that $[x] \in \text{dom}(h)$, $h_s(x) = h([x])$ holds. We claim that for every $s \in \{0, \ldots, n\}$, such a homomorphism h_s consistent with h exists. Then $h' = h_n$ is the desired homomorphism. This claim can be proved by induction on s.

In order to actually construct the homomorphism $h' = h_n$, we may thus simply replay the chase and construct h_s for every $s \in \{0, \ldots, n\}$. The length n of the chase is polynomially bounded (cf. Section 2.1). The action required to construct h_s from h_{s-1} fits into polynomial time as well. We thus get the desired upper bound on the time needed for the construction of h'. □

The only ingredient missing for our FindCore^E algorithm is an efficient search for a homomorphism $h\colon T_{xy} \to U$ with $U \subseteq T^{\Sigma_t}$.

Procedure FindCore^{E}

Input: Source ground instance S
Output: Core of a universal solution for S

(1) Chase (S,∅) with Σ_{st} to obtain $(S,T) := (S,\emptyset)^{\Sigma_{st}}$;
(2) Chase T with Σ_t to obtain $U := T^{\Sigma_t}$;
(3) **for** each $x \in \mathrm{var}(U)$, $y \in \mathrm{dom}(U)$, $x \neq y$ **do**
(4) | Compute T_{xy};
(5) | Look for $h\colon T_{xy} \to U$ s.t. $h(x) = h(y)$;
(6) | **if** there is such h **then**
(7) | | Extend h to an endomorphism h' on U;
(8) | | Transform h' into a retraction r;
(9) | | Set $U := r(U)$;
(10) **return** U.

By the construction of T_{xy} according to Lemma 3, the domain size of T_{xy} as well as the number of facts in it are only by a constant larger than those of the corresponding preuniversal instance T. By Theorem 1, the complexity of searching for a homomorphism is determined by the block size. The problem with EGDs in the target chase is that they may destroy the block structure of T by equating variables from different blocks of T. However, we show below that the search for a homomorphism on T_{xy} may still use the blocks of $T^{\Sigma_{st}}$ computed *before* the target chase. To achieve this, we adapt the *Rigidity Lemma* from [3]. The original *Rigidity Lemma* was formulated for sets of target dependencies consisting of EGDs only. A close inspection of the proof in [3] reveals that it remains valid when TGDs are added.

Definition 1. *Let K be an instance whose elements are constants and nulls. Let y be some element of K. We say that y is* rigid *if $h(y) = y$ for every endomorphism h on K. In particular, all constants of K are rigid.*

Lemma 4. *(*Rigidity*) Assume a data exchange setting where Σ_{st} is a set of TGDs and Σ_t is a set of EGDs* and TGDs. *Let J be the canonical preuniversal instance and let $J' = J^{\Sigma_t}$ be the canonical universal instance. Let x and y be nulls of J s.t. $x \backsim y$ (i.e., $[x] = [y]$) and s.t. $[x]$ is a nonrigid null of J'. Then x and y are in the same block of J.*

Next, we formalize the idea of considering the blocks of the preuniversal instance when searching for a homomorphism on the universal instance.

Definition 2. *We define the* non-rigid Gaifman graph $\mathcal{G}'(I)$ *of an instance I as the usual Gaifman graph but restricted to vertices corresponding to non-rigid variables. We define* non-rigid blocks *of an instance I as the connected components of the non-rigid Gaifman graph $\mathcal{G}'(I)$.*

Theorem 7. *Let T be a preuniversal instance obtained via the STDs Σ_{st}. Let Σ_t be a set of weakly acyclic TGDs and EGDs, and let U be a retract of T^{Σ_t}. Moreover, let $x, y \in \mathrm{dom}(T^{\Sigma_t})$ and let $T_{xy} \subseteq T^{\Sigma_t}$ be constructed according to Lemma 3. Then we can check if there exists a homomorphism $h\colon T_{xy} \to U$, s.t. $h(x) = h(y)$ in time $O(|dom(U)|^c)$ for some c depending only on $\Sigma = \Sigma_{st} \cup \Sigma_t$.*

Proof. First, it can be easily shown that the rigid variables of T^{Σ_t} are also rigid in T_{xy}. The key observation to achieve the $O(|dom(U)|^c)$ upper bound on the complexity is that the search for a homomorphism $h\colon T_{xy} \to U$ proceeds by inspecting the non-rigid blocks of T_{xy} individually. Moreover, since we already

have a retraction $r : T^{\Sigma} \to U$, we may search for a homomorphism h with $h(x) = h(y)$ by inspecting only the blocks containing x and y and to set $h(z) = r(z)$ for the variables of all other blocks. □

Putting all these pieces together, we get the FindCore^E algorithm. It has basically the same overall structure as the FindCore algorithm of [6], which we recalled in Section 2.2. Of course, the correctness of our algorithm and its polynomial time upper bound are now based on the new results proved in this section. In particular, step (4) is based on Lemma 3, step (5) is based on Lemma 4 and Theorem 7, and step (7) is based on Theorem 6.

Theorem 8. *Let* $(\mathbf{S}, \mathbf{T}, \Sigma_{st}, \Sigma_t)$ *be a data exchange setting with STDs* Σ_{st} *and TDs* Σ_t*. Moreover, let* S *be a ground instance of the target schema* $\mathbf{S}$*. If this data exchange problem has a solution, then* FindCore^E *correctly computes the core of a canonical universal solution in time* $O(|\mathrm{dom}(S)|^b)$ *for some* b *that depends only on* $\Sigma_{st} \cup \Sigma_t$*.*

4 Implementation and Experimental Results

Implementation. We have implemented the FindCore^E algorithm presented in Section 3 in a prototype system. Its principal architecture is shown in Figure 3(a). For specifying data exchange scenarios, we use XML configuration files. The schema of the source and target database as well as the STDs and TDs are thus cleanly separated from the scenario-independent Java code. The XML configuration data is passed to the Java program, which uses XSLT templates to automatically generate those code parts which depend on the concrete scenario – in particular, the SQL-statements for managing the target database (creating tables and views, transferring data between tables etc.).

None of the common DBMSs to-date support labeled nulls. Therefore, to implement this feature, we had to augment every target relation (i.e., table) with additional columns, storing null labels. For instance, for a column `tutor` of the `Tutor` table, a column `tutor_var` is created to store the labels for nulls of `tutor`. To simulate homomorphisms, we use a table called `Map` storing variable mappings, and views that substitute labeled nulls in the data tables with their

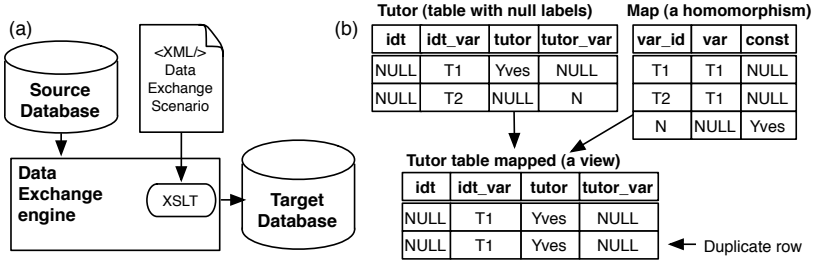


Fig. 3. Overview of the implementation (a) and modelling labeled nulls (b)

images given by a homomorphism. Figure 3(b) gives a flavor of what this part of the database looks like. The target database contains many more auxiliary tables for maintaining the relevant information of the core computation.

A great deal of the core computation is delegated to the target DBMS via SQL commands. For instance, the homomorphism computation in step 5 of FindCore^E is performed in the following way. Let a variable x and a term y be selected at step 3 of the algorithm, and let the set T_{xy} be computed at step 4. We want to build a homomorphism $h\colon T_{xy} \to U$, s.t. $h(x) = h(y)$. To do so, we need to inspect all possible mappings from the block of x and from the block of y. Each of these steps boils down to generating and executing a database query that fetches all possible substitutions for the variables in each block.

Example 5. Let us revisit the data exchange setting from Example 1. Suppose that the canonical universal solution is

$J = \{$`Course(`C_1`,'java'), Tutor(`T_2`,`N`), Teaches(`T_2`,`C_1`), NeedsLab(`T_2`,`L_2`),`
`Course(`C_2`,'java'), Tutor(`T_1`,'Yves'), Teaches(`T_1`,`C_2`), NeedsLab(`T_1`,`L_1`)`$\}$

Suppose that we look for a proper endomorphism h' on J and suppose that step 4 of FindCore^E yields the set $T_{N,'\mathsf{Yves}'} = \{$`Tutor(`$T_2$`,`$N$`),Teaches(`$T_2$`,`$C_1$`), Course(`$C_1$`,'java')`$\}$. At step 5, a homomorphism $h\colon T_{xy} \to J$ (with $x = N$ and $y =' \mathsf{Yves}'$), s.t. $h(N) = '\mathsf{Yves}'$ has to be found. In the absence of EGDs, non-rigid blocks are the same as usual blocks, and the block of N in $T_{N,'\mathsf{Yves}'}$ is $\{N, T_2, C_1\}$. The following SQL query returns all possible instantiations of the variables $\{T_2, C_1\}$ compatible with the mapping $h(N) = '\mathsf{Yves}'$:

SELECT Tutor.idt_var **AS** T2, Course.idc_var **AS** C1 **FROM** Tutor **JOIN** Teaches **ON** Tutor.idt_var = Teaches.id_tutor_var **JOIN** Course **ON** Teaches.id_course_var = Course.idc_var **WHERE** Tutor.tutor='Yves' **AND** Course.course='java'

In our example, the result is $\{T_2 \leftarrow T_1,\ C_1 \leftarrow C_2\}$.

Experiments. We have run experiments with our prototype implementation on several scenarios with varying size of the schema (5–10 target relations), of the dependencies (5–15 constraints), and of the actual source data. Since there

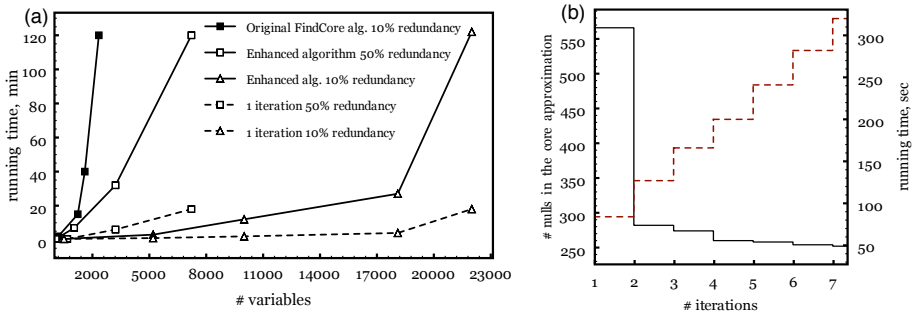


Fig. 4. Performance (a) and the progress (b) of core computation

are no established benchmarks for core computation algorithms, we constructed our own test cases by appropriately extending the data exchange scenario from Example 1. Typical runtimes are displayed in Figure 4. They were obtained by tests on a workstation running Suse Linux with 2 QuadCore processors (2.3 GHz) and 16 GB RAM. Oracle 11g was used as database system.

We had to synthesize the scenarios ourselves since no benchmark for core computation exists currently, and we needed means to adjust the dependencies and the source data in order to manage redundancy in the target database. When the target EGDs were deliberately "badly designed", the canonical solution had about 50% more nulls than the core. In this case, our system handled only about 7,000 nulls in the target DB in 120 min (2nd solid curve from the left). In contrast, when the target EGDs were "carefully designed", the canonical solution had only 10% more nulls than the core. In this case, about 22,000 nulls were handled in similar time (3rd solid curve).

We have also implemented the FindCore algorithm of [6] in order to compare its performance with our algorithm. The left-most curve in Figure 4(a) corresponds to a run of FindCore on the "well-designed" data exchange problem. The runtime is comparable to FindCore^E in case of "badly designed" dependencies. Actually, this is not surprising: One of the principal advantages of the FindCore^E algorithm is that it enforces EGDs as part of the chase rather than in the course of the core computation. The negative effect of simulating the EGDs by TGDs is illustrated by the following simple example:

Example 6. Let $J = \{R(x,y), P(y,x)\}$ be a preuniversal instance, and let a single EGD $R(z,v), P(v,z) \rightarrow z = v$ constitute Σ_t. To simulate this EGD by TGDs in [6], the following set of dependencies $\bar{\Sigma}_t$ has to be constructed:

$$
\begin{array}{llll}
P(x,y) \rightarrow E(x,x) & E(x,y) \rightarrow E(y,x) & & \\
P(x,y) \rightarrow E(y,y) & E(x,y), E(y,z) \rightarrow E(x,z) & P(x,y), E(x,z) \rightarrow P(z,y) & \\
R(x,y) \rightarrow E(x,x) & R(x,y), E(x,z) \rightarrow R(z,y) & P(x,y), E(y,z) \rightarrow P(x,z) & \\
R(x,y) \rightarrow E(y,y) & R(x,y), E(y,z) \rightarrow R(x,z) & R(z,v), P(v,z) \rightarrow E(z,v) &
\end{array}
$$

where E is the auxiliary predicate representing equality. Chasing J with $\bar{\Sigma}_t$ (in a nice order), yields the instance $J^{\bar{\Sigma}_t} = \{R(x,y), R(x,x), R(y,x), R(y,y), P(y,x), P(y,y), P(x,y), P(x,x), E(x,x), E(x,y), E(y,x), E(y,y)\}$. The core computation applied to $J^{\bar{\Sigma}_t}$ produces the instance $\{R(x,x), P(x,x)\}$ or $\{R(y,y), P(y,y)\}$. On the other hand, if EGDs were directly enforced by the target chase, then the chase would end with the canonical universal solution $J^{\Sigma_t} = \{R(x,x), P(x,x)\}$.

Another interesting observation is that, in many cases, the result of applying just a few endomorphisms already leads to a significant elimination of *redundant* nulls (i.e., nulls present in the canonical solution but not in the core) from the target database and that further iterations of this procedure are much less effective concerning the number of nulls eliminated vs. time required. A typical situation is shown in Figure 4(b): The solid line shows the number of redundant nulls remaining after i iterations (i.e., i nested endomorphisms) while the dotted line shows the total time required for the first i iterations. To achieve this, we used several heuristics to choose the best homomorphisms. The following hints

proved quite useful: (i) prefer constants over variables, (ii) prefer terms already used as substitutions, and (iii) avoid mapping a variable onto itself.

Every intermediate database instance produced by FindCore^E is a universal solution to the data exchange problem. Hence, our prototype implementation also allows the user to restrict the number of nested endomorphisms to be constructed, thus computing an approximation of the core rather than the core itself. The dotted curves in Figure 4(a) corresponds to a "partial" core computation, with only 1 iteration of the while-loop in FindCore^E. In both scenarios, even a single endomorphism allowed us to eliminate over 85% of all redundant nulls.

Lessons learned. Our experiments have clearly revealed the importance of carefully designing target EGDs. In some sense, they play a similar role as the core computation in that they lead to an elimination of nulls. However, the EGDs do it much more efficiently. Another observation is that it is well worth considering to content oneself with an approximation of the core since, in general, a small number of iterations of our algorithm already leads to a significant reduction of nulls. Finally, the experience gained with our experiments gives us several hints for future performance improvements. We just give three examples: `(i)` Above all, further heuristics have to be incorporated concerning the search for an endomorphism which maps a labeled null onto some other domain element. So far, we have identified and implemented only the most straightforward, yet quite effective, rules. Apparently, additional measures are needed to further prune the search space. `(ii)` We have already mentioned the potential of approximating the core by a small number of endomorphisms. Again, we need further heuristics concerning the search for the most effective endomorphisms. `(iii)` Some phases of the search for an endomorphism allow for concurrent implementation. This potential of parallelization, which has not been exploited so far, clearly has to be leveraged in future versions of our implementation.

5 Conclusion

In this paper we have revisited the core computation in data exchange and we have come up with an enhanced version of the FindCore algorithm from [6], which avoids the simulation of EGDs by TGDs. The algorithms FindCore and FindCore^E look similar in structure and have essentially the same asymptotic worst-case behavior. More precisely, both algorithms are exponential w.r.t. some constant b which depends on the dependencies $\Sigma_{st} \cup \Sigma_t$ of the data exchange setting. Actually, in [9] it was shown that the core computation for a given target instance J is fixed-parameter intractable w.r.t. its block size. Hence, a significant reduction of the worst-case complexity is not likely to be achievable. At any rate, as we have discussed in Section 4, our new approach clearly outperforms the previous one under realistic assumptions.

We have also presented a prototype implementation of our algorithm, which delegates most of its work to the underlying DBMS via SQL. It has thus been demonstrated that core computation fits well into existing database technology and is clearly not a separate technology. Although the data exchange scenarios

tackled so far are not industrial size examples, we expect that there is ample space for performance improvements. The experience gained with our prototype gives valuable hints for directions of future work.

References

1. Fagin, R.: Horn clauses and database dependencies. J. ACM 29, 952–985 (1982)
2. Fagin, R., Kolaitis, P.G., Miller, R.J., Popa, L.: Data exchange: semantics and query answering. Theor. Comput. Sci. 336, 89–124 (2005)
3. Fagin, R., Kolaitis, P.G., Popa, L.: Data exchange: getting to the core. ACM Trans. Database Syst. 30, 174–210 (2005)
4. Beeri, C., Vardi, M.Y.: A proof procedure for data dependencies. J. ACM 31, 718–741 (1984)
5. Hell, P., Nešetřil, J.: The core of a graph. Discrete Mathematics 109, 117–126 (1992)
6. Gottlob, G., Nash, A.: Data exchange: computing cores in polynomial time. In: Proc. PODS 2006, pp. 40–49. ACM Press, New York (2006)
7. Pichler, R., Savenkov, V.: Towards practical feasibility of core computation in data exchange. Technical Report DBAI-TR-2008-57, TU Vienna (2008), http://www.dbai.tuwien.ac.at/research/report/dbai-tr-2008-57.pdf
8. Deutsch, A., Tannen, V.: Reformulation of XML queries and constraints. In: Calvanese, D., Lenzerini, M., Motwani, R. (eds.) ICDT 2003. LNCS, vol. 2572, pp. 225–238. Springer, Heidelberg (2002)
9. Gottlob, G.: Computing cores for data exchange: new algorithms and practical solutions. In: Proc. PODS 2005, pp. 148–159. ACM Press, New York (2005)

Data-Oblivious Stream Productivity

Jörg Endrullis[1], Clemens Grabmayer[2], and Dimitri Hendriks[1]

[1] Vrije Universiteit Amsterdam, Department of Computer Science
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands
joerg@few.vu.nl, diem@cs.vu.nl
[2] Universiteit Utrecht, Department of Philosophy
Heidelberglaan 8, 3584 CS Utrecht, The Netherlands
clemens@phil.uu.nl

Abstract. We are concerned with demonstrating productivity of specifications of infinite streams of data, based on orthogonal rewrite rules. In general, this property is undecidable, but for restricted formats computable sufficient conditions can be obtained. The usual analysis, also adopted here, disregards the identity of data, thus leading to approaches that we call data-oblivious. We present a method that is provably optimal among all such data-oblivious approaches. This means that in order to improve on our algorithm one has to proceed in a data-aware fashion.[1]

1 Introduction

For programming with infinite structures, productivity is what termination is for programming with finite structures. Productivity captures the intuitive notion of unlimited progress, of 'working' programs producing defined values indefinitely. In functional languages, usage of infinite structures is common practice. For the correctness of programs dealing with such structures one must guarantee that every finite part of the infinite structure can be evaluated, that is, the specification of the infinite structure must be productive.

We investigate this notion for stream specifications, formalized as orthogonal term rewriting systems. Common to all previous approaches for recognizing productivity is a quantitative analysis that abstracts away from the concrete values of stream elements. We formalize this by a notion of 'data-oblivious' rewriting, and introduce the concept of data-oblivious productivity. Data-oblivious (non-)productivity implies

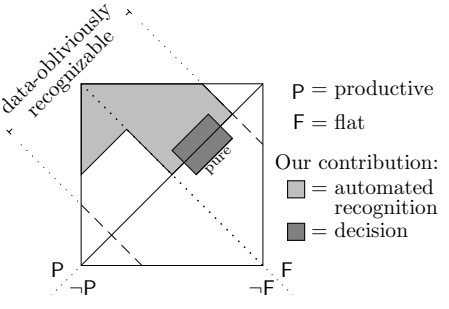


Fig. 1. Map of stream specifications

[1] This research has been partially funded by the Netherlands Organisation for Scientific Research (NWO) under FOCUS/BRICKS grant number 642.000.502.

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 79–96, 2008.

(non-)productivity, but neither of the converse implications holds. Fig. 1 shows a Venn diagram of stream specifications, highlighting the subset of 'data-obliviously recognizable' specifications where (non-)productivity can be recognized by a data-oblivious analysis.

We identify two syntactical classes of stream specifications: 'flat' and 'pure' specifications, see the description below. For the first we devise a decision algorithm for data-oblivious (d-o) productivity. This gives rise to a computable, d-o optimal, criterion for productivity: every flat stream specification that can be established to be productive by whatever d-o argument is recognized as productive by this criterion (see Fig. 1). For the subclass of pure specifications, we establish that d-o productivity coincides with productivity, and thereby obtain a decision algorithm for productivity of this class. Additionally, we extend our criterion beyond the class of flat stream specifications, allowing for 'friendly nesting' in the specification of stream functions; here d-o optimality is not preserved.

In defining the different formats of stream specifications, we distinguish between rules for stream constants, and rules for stream functions. Only the latter are subjected to syntactic restrictions. In flat stream specifications the defining rules for the stream functions do not have nesting of stream function symbols; however, in defining rules for stream constants nesting of stream function symbols *is* allowed. This format makes use of exhaustive pattern matching on data to define stream functions, allowing for multiple defining rules for an individual stream function symbol. Since the quantitative consumption/production behaviour of a symbol f might differ among its defining rules, in a d-o analysis one has to settle for the use of lower bounds when trying to recognize productivity. If for all stream function symbols f in a flat specification $\mathcal{T}$ the defining rules for f coincide, disregarding the identity of data-elements, then $\mathcal{T}$ is called pure.

Our decision algorithm for d-o productivity determines the tight d-o lower bound on the production behaviour of every stream function, and uses these bounds to calculate the d-o production of stream constants. We briefly explain both aspects. Consider the stream specification $\mathsf{A} \to \mathsf{0} : \mathsf{f}(\mathsf{A})$ together with the rules $\mathsf{f}(\mathsf{0} : \sigma) \to \mathsf{1} : \mathsf{0} : \mathsf{1} : \mathsf{f}(\sigma)$, and $\mathsf{f}(\mathsf{1} : \sigma) \to \mathsf{0} : \mathsf{f}(\sigma)$, defining the stream $\mathsf{0} : \mathsf{1} : \mathsf{0} : \mathsf{1} : \ldots$ of alternating bits. The tight d-o lower bound for f is the function $id\colon n \mapsto n$. Further note that $suc\colon n \mapsto n+1$ captures the quantitative behaviour of the function prepending a data element to a stream term. Therefore the d-o production of A can be computed as $\mathrm{lfp}(suc \circ id) = \infty$, where $\mathrm{lfp}(f)$ is the least fixed point of $f : \overline{\mathbb{N}} \to \overline{\mathbb{N}}$ and $\overline{\mathbb{N}} := \mathbb{N} \cup \{\infty\}$; hence A is productive. As a comparison, only a 'data-aware' approach is able to establish productivity of $\mathsf{B} \to \mathsf{0} : \mathsf{g}(\mathsf{B})$ with $\mathsf{g}(\mathsf{0} : \sigma) \to \mathsf{1} : \mathsf{0} : \mathsf{g}(\sigma)$, and $\mathsf{g}(\mathsf{1} : \sigma) \to \mathsf{g}(\sigma)$. The d-o lower bound of g is $n \mapsto 0$, due to the latter rule. This makes it impossible for any conceivable d-o approach to recognize productivity of B.

We obtain the following results:

(i) For the class of flat stream specifications we give a computable, d-o optimal, sufficient condition for productivity.
(ii) We show decidability of productivity for the class of pure stream specifications, an extension of the format in [3].

(iii) Disregarding d-o optimality, we extend (i) to the bigger class of friendly nesting stream specifications.
(iv) A tool automating (i), (ii) and (iii), which can be downloaded from, and used via a web interface at: `http://infinity.few.vu.nl/productivity`.

Related work. Previous approaches [6,4,7,1] employed d-o reasoning (without using this name for it) to find sufficient criteria ensuring productivity, but did not aim at optimality. The d-o production behaviour of a stream function f is bounded from below by a 'modulus of production' $\nu_{\mathsf{f}} : \mathbb{N}^k \to \mathbb{N}$ with the property that the first $\nu_{\mathsf{f}}(n_1, \ldots, n_k)$ elements of $\mathsf{f}(t_1, \ldots, t_k)$ can be computed whenever the first n_i elements of t_i are defined. Sijtsma develops an approach allowing arbitrary production moduli $\nu : \mathbb{N}^k \to \mathbb{N}$, which, while providing an adequate mathematical description, are less amenable to automation. Telford and Turner [7] employ production moduli of the form $\nu(n) = n + a$ with $a \in \mathbb{Z}$. Hughes, Pareto and Sabry [4] use $\nu(n) = \max\{c \cdot x + d \mid x \in \mathbb{N},\ n \geq a \cdot x + b\} \cup \{0\}$ with $a, b, c, d \in \mathbb{N}$. Both classes of production moduli are strictly contained in the class of 'periodically increasing' functions which we employ in our analysis. We show that the set of d-o lower bounds of flat stream function specifications is exactly the set of periodically increasing functions. Buchholz [1] presents a type system for productivity, using unrestricted production moduli. For a restricted subclass he gives an automatable method for ensuring productivity, but this excludes the use of stream functions with a negative effect like odd defined by $\mathsf{odd}(x{:}y{:}\sigma) \to y{:}\mathsf{odd}(\sigma)$ with a (periodically increasing) modulus $\nu_{\mathsf{odd}}(n) = \lfloor \frac{n}{2} \rfloor$.

Overview. In Sec. 2 we define the notion of stream specification, and the syntactic format of flat and pure specifications. In Sec. 3 we formalize the notion of d-o rewriting. In Sec. 4 we introduce a production calculus as a means to compute the production of the data-abstracted stream specifications, based on the set of periodically increasing functions. A translation of stream specifications into production terms is defined in Sec. 5. Our main results, mentioned above, are collected in Sec. 6. We conclude and discuss some future research topics in Sec. 7.

2 Stream Specifications

We introduce the notion of stream specification. An example is given in Fig. 2, a productive specification of Pascal's triangle where the rows are separated by zeros. Indeed, evaluating this specification, we get: $\mathsf{P} \twoheadrightarrow 0:1:0:1:1:0:1:2:1:0:1:3:3:1:\ldots$.

$$\begin{array}{c} \mathsf{P} \to 0 : \mathsf{s}(0) : \mathsf{f}(\mathsf{P}) \\ \mathsf{f}(\mathsf{s}(x) : y : \sigma) \to \mathsf{a}(\mathsf{s}(x), y) : \mathsf{f}(y : \sigma) \\ \mathsf{f}(0 : \sigma) \to 0 : \mathsf{s}(0) : \mathsf{f}(\sigma) \\ \hline \mathsf{a}(\mathsf{s}(x), y) \to \mathsf{s}(\mathsf{a}(x, y)) \\ \mathsf{a}(0, y) \to y \end{array}$$

Fig. 2. A flat stream specification

We define stream specifications to consist of a *stream layer* (top) where stream constants and functions are specified, and a *data layer* (bottom) such that the stream layer may use symbols of the data layer, but not vice-versa. Thus, the data layer is a term

rewriting system on its own. In order to abstract from the termination problem when investigating productivity, we require the data layer to be strongly normalizing. Let us explain the reason for this hierarchical setup. Stream dependent data symbols (whose defining rules do contain stream symbols), like $\mathsf{head}(x : \sigma) \to x$, might cause the output of undefined data terms. Let $\sigma(n) := \mathsf{head}(\mathsf{tail}^n(\sigma))$, and consider the following bitstream specifications:

$$\mathsf{S} \to 0 : \mathsf{S}(2) : \mathsf{S} \qquad\qquad \mathsf{T} \to 0 : \mathsf{T}(3) : \mathsf{T}\,,$$

taken from [6]. Here we have that $\mathsf{S}(n) \to^* \mathsf{S}(n-2)$ for all $n \geq 2$, and $\mathsf{S}(1) \to^* \mathsf{S}(2)$, and hence $\mathsf{S} \twoheadrightarrow 0 : 0 : 0 : \ldots$, producing the infinite stream of zeros. On the other hand, the evaluation of each data term $\mathsf{T}(2n+1)$ eventually ends up in the loop $\mathsf{T}(3) \to^* \mathsf{T}(1) \to^* \mathsf{T}(3) \to^* \ldots$. Hence we have that $\mathsf{T} \twoheadrightarrow 0 : ? : 0 : ? : \ldots$ (where ? stands for 'undefined') and T is not productive.

Such examples, where the evaluation of stream elements needs to be delayed to wait for 'future information', can only be productive using a lazy evaluation strategy like in the programming language Haskell. Productivity of specifications like these is adequately analyzed using the concept of 'set productivity' in [6]. A natural first step is to study its proper subclass called 'segment productivity', where well-definedness of one element requires well-definedness of all previous ones. The restriction to this subclass is achieved by disallowing stream dependent data functions. While conceptually more general, in practice stream dependent data functions usually can be replaced by pattern matching: $\mathsf{add}(\sigma, \tau) \to (\mathsf{head}(\sigma) + \mathsf{head}(\tau)) : \mathsf{add}(\mathsf{tail}(\sigma), \mathsf{tail}(\tau))$, for example, can be replaced by the better readable $\mathsf{add}(x : \sigma, y : \tau) \to (x + y) : \mathsf{add}(\sigma, \tau)$.

Stream specifications are formalized as many-sorted, orthogonal, constructor term rewriting systems [8]. We distinguish between *stream terms* and *data terms*. For the sake of simplicity we consider only one sort S for stream terms and one sort D for data terms. Without any complication, our results extend to stream specifications with multiple sorts for data terms and for stream terms.

Let U be a finite set of *sorts*. A *U-sorted set* A is a family of sets $\{A_u\}_{u \in U}$; for $V \subseteq U$ we define $A_V := \bigcup_{v \in V} A_v$. A *$U$-sorted signature* Σ is a U-sorted set of function symbols f, each equipped with an arity $ar(f) = \langle u_1 \cdots u_n, u \rangle \in U^* \times U$ where u is the sort of f; we write $u_1 \times \ldots \times u_n \to u$ for $\langle u_1 \cdots u_n, u \rangle$. Let X be a U-sorted set of variables. The *U-sorted set of terms* $Term(\Sigma, X)$ is inductively defined by: for all $u \in U$, $X_u \subseteq Term(\Sigma, X)_u$, and $f(t_1, \ldots, t_n) \in Term(\Sigma, X)_u$ if $f \in \Sigma$, $ar(f) = u_1 \times \ldots \times u_n \to u$, and $t_i \in Term(\Sigma, X)_{u_i}$. $Term_\infty(\Sigma, X)$ denotes the set of *(possibly) infinite terms over Σ and X* (see [8]). Usually we keep the set of variables implicit and write $Term(\Sigma)$ and $Term_\infty(\Sigma)$. A *U-sorted term rewriting system (TRS)* is a pair $\langle \Sigma, R \rangle$ consisting of a U-sorted signature Σ and a U-sorted set R of *rules* that satisfy well-sortedness, for all $u \in U$: $R_u \subseteq Term(\Sigma, X)_u \times Term(\Sigma, X)_u$, as well as the standard TRS requirements.

Let $\mathcal{T} = \langle \Sigma, R \rangle$ be a U-sorted TRS. For a term $t \in Term(\Sigma)_u$ where $u \in U$ we denote the root symbol of t by $root(t)$. We say that two occurrences of symbols in a term are *nested* if the position [8, p.29] of one is a prefix of the position of the other. We define $\mathcal{D}(\Sigma) := \{root(l) \mid l \to r \in R\}$, the set of *defined symbols*, and

$\mathcal{C}(\Sigma) := \Sigma \setminus \mathcal{D}(\Sigma)$, the set of *constructor symbols*. Then $\mathcal{T}$ is called a *constructor TRS* if for every rewrite rule $\rho \in R$, the left-hand side is of the form $f(t_1, \ldots, t_n)$ with $t_i \in \mathit{Term}(\mathcal{C}(\Sigma))$; then ρ is called a *defining rule for* f. We call $\mathcal{T}$ *exhaustive* for $f \in \Sigma$ if every term $f(t_1, \ldots, t_n)$ with (possibly infinite) closed constructor terms t_i is a redex. Note that, stream constructor terms are inherently infinite.

A *stream TRS* is a finite $\{S, D\}$-sorted, orthogonal, constructor TRS $\langle \Sigma, R \rangle$ such that ':' $\in \Sigma_S$, the *stream constructor symbol*, with arity $D \times S \to S$ is the single constructor symbol in Σ_S. Elements of Σ_D and Σ_S are called the *data symbols* and the *stream symbols*, respectively. We let $\Sigma_S^- := \Sigma_S \setminus \{':'\}$, and, for all $\mathsf{f} \in \Sigma_S^-$, we assume, without loss of generality, that the stream arguments are in front: $ar(\mathsf{f}) \in S^{ar_s(\mathsf{f})} \times D^{ar_d(\mathsf{f})} \to S$, where $ar_s(\mathsf{f})$ and $ar_d(\mathsf{f}) \in \mathbb{N}$ are called the *stream arity* and the *data arity* of f, respectively. By Σ_{scon} we denote the set of symbols in Σ_S^- with stream arity 0, called the *stream constant symbols*, and $\Sigma_{sfun} := \Sigma_S^- \setminus \Sigma_{scon}$ the set of symbols in Σ_S^- with stream arity unequal to 0, called the *stream function symbols*. Note that stream constants may have a data arity > 0 as for example in: $\mathsf{natsFrom}(n) \to n : \mathsf{natsFrom}(\mathsf{s}(n))$. Finally, by R_{scon} we mean the defining rules for the symbols in Σ_{scon}.

Definition 2.1. A *stream specification* $\mathcal{T}$ is a stream TRS $\mathcal{T} = \langle \Sigma, R \rangle$ such that the following conditions hold:

(i) There is a designated symbol $\mathsf{M}_0 \in \Sigma_{scon}$ with $ar_d(\mathsf{M}_0) = 0$, the *root of* $\mathcal{T}$.
(ii) $\langle \Sigma_D, R_D \rangle$ is a terminating, D-sorted TRS; R_D is called the *data layer* of $\mathcal{T}$.
(iii) $\mathcal{T}$ is exhaustive (for all defined symbols in $\Sigma = \Sigma_S \uplus \Sigma_D$).

Note that Def. 2.1 indeed imposes a hierarchical setup; in particular, stream dependent data functions are excluded by item (ii). Exhaustivity for Σ_D together with strong normalization of R_D guarantees that closed data terms rewrite to constructor normal forms, a property known as sufficient completeness [5].

We are interested in productivity of recursive stream specifications that make use of a library of 'manageable' stream functions. By this we mean a class of stream functions defined by a syntactic format with the property that their d-o lower bounds are computable and contained in a set of production moduli that is effectively closed under composition, pointwise infimum and where least fixed points can be computed. As such a format we define the class of flat stream specifications (Def. 2.2) for which d-o lower bounds are precisely the set of 'periodically increasing' functions (see Sec. 4). Thus only the stream function rules are subject to syntactic restrictions. No condition other than well-sortedness is imposed on the defining rules of stream constant symbols.

In the sequel let $\mathcal{T} = \langle \Sigma, R \rangle$ be a stream specification. We define the relation $\rightsquigarrow$ on rules in R_S: for all $\rho_1, \rho_2 \in R_S$, $\rho_1 \rightsquigarrow \rho_2$ (ρ_1 *depends on* ρ_2) holds if and only if ρ_2 is the defining rule of a stream function symbol on the right-hand side of ρ_1. Furthermore, for a binary relation $\to \subseteq A \times A$ on a set A we define $(a \to) := \{b \in A \mid a \to b\}$ for all $a \in A$, and we denote by $\to^+$ and $\to^*$ the *transitive closure* and the *reflexive–transitive closure* of $\to$, respectively.

Definition 2.2. A rule $\rho \in R_S$ is called *nesting* if its right-hand side contains nested occurrences of stream symbols from Σ_S^-. We use R_{nest} to denote the subset of nesting rules of R and define $R_{\neg nest} := R_S \setminus R_{nest}$, the set of *non-nesting rules*.

A rule $\rho \in R_S$ is called *flat* if all rules in $(\rho \rightsquigarrow^*)$ are non-nesting. A symbol $\mathsf{f} \in \Sigma_S^-$ is called *flat* if all defining rules of f are flat; the set of flat symbols is denoted Σ_{flat}. A stream specification $\mathcal{T}$ is called *flat* if $\Sigma_S^- \subseteq \Sigma_{flat} \cup \Sigma_{scon}$, that is, all symbols in Σ_S^- are either flat or stream constant symbols.

See Fig. 2 and Ex. 5.5 for examples of flat stream specifications.

As the basis of d-o rewriting (see Def. 3.2) we define the data abstraction of terms as the results of replacing all data-subterms by the symbol $\bullet$.

Definition 2.3. Let $(\!|\Sigma|\!) := \{\bullet\} \uplus \Sigma_S$. For stream terms $s \in \mathit{Term}(\Sigma)_S$, the *data abstraction* $(\!|s|\!) \in \mathit{Term}((\!|\Sigma|\!))_S$ is defined by:

$$(\!|\sigma|\!) = \sigma \quad (\!|u : s|\!) = \bullet : (\!|s|\!) \quad (\!|\mathsf{f}(s_1, \ldots, s_n, u_1, \ldots, u_m)|\!) = \mathsf{f}((\!|s_1|\!), \ldots, (\!|s_n|\!), \bullet, \ldots, \bullet).$$

Based on this definition of data abstracted terms, we define the class of pure stream specifications, an extension of the equally named class in [3].

Definition 2.4. A stream specification $\mathcal{T}$ is called *pure* if it is flat and if for every symbol $\mathsf{f} \in \Sigma_S^-$ the data abstractions $(\!|\ell|\!) \to (\!|r|\!)$ of the defining rules $\ell \to r$ of f coincide (modulo renaming of variables).

See Ex. 5.4 for an example of a pure stream function specification. Def. 2.4 generalizes the specifications called 'pure' in [3] in four ways concerning the defining rules of stream functions: First, the requirement of right-linearity of stream variables is dropped, allowing for rules like $\mathsf{f}(\sigma) \to \mathsf{g}(\sigma, \sigma)$. Second, 'additional supply' to the stream arguments is allowed. For instance, in a rule like $\mathsf{diff}(x : y : \sigma) \to \mathsf{xor}(x, y) : \mathsf{diff}(y : \sigma)$, the variable y is 'supplied' to the recursive call of diff. Third, the use of non-productive stream functions is allowed now, relaxing an earlier requirement of [3] on stream function symbols to be 'weakly guarded', see Def. 5.1. Finally, defining rules for stream function symbols may use a restricted form of pattern matching as long as, for every stream function f, the d-o consumption/production behaviour (see Sec. 3) of all defining rules for f is the same.

Definition 2.5. A rule $\rho \in R_S$ is called *friendly* if for all rules $\gamma \in (\rho \rightsquigarrow^*)$ we have: (1) γ consumes in each argument at most one stream element, and (2) it produces at least one. The set of *friendly nesting rules* R_{fnest} is the largest extension of the set of friendly rules by non-nesting rules from R_S that is closed under $\rightsquigarrow$. A symbol $\mathsf{f} \in \Sigma_S^-$ is *friendly nesting* if all defining rules of f are friendly nesting. A stream specification $\mathcal{T}$ is called *friendly nesting* if $\Sigma_S^- \subseteq \Sigma_{fnest} \cup \Sigma_{scon}$, that is, all symbols in Σ_S^- are either friendly nesting or stream constant symbols.

Note that, in particular, every flat stream specification is friendly nesting.

Example 2.6. The rules $\mathsf{X} \to 0 : \mathsf{f}(\mathsf{X})$ and $\mathsf{f}(x : \sigma) \to x : \mathsf{f}(\mathsf{f}(\sigma))$ form a friendly nesting stream specification with an empty data layer.

Definition 2.7. Let $\mathcal{A} = \langle Term(\Sigma)_S, \to \rangle$ be an abstract reduction system (ARS) on the set of terms over a stream TRS signature Σ. The *production function* $\Pi_{\mathcal{A}} : Term(\Sigma)_S \to \overline{\mathbb{N}}$ of $\mathcal{A}$ is defined for all $s \in Term(\Sigma)_S$ by:

$$\Pi_{\mathcal{A}}(s) := \sup \{\, n \in \mathbb{N} \mid s \to^*_{\mathcal{A}} u_1 : \ldots : u_n : t \,\} .$$

We call $\mathcal{A}$ *productive for a stream term* s if $\Pi_{\mathcal{A}}(s) = \infty$. A stream specification $\mathcal{T}$ is called *productive* if $\mathcal{T}$ is productive for its root M_0.

Note that in a stream specification $\mathcal{T}$ it holds (since $\mathcal{T}$ is an orthogonal rewriting system) that if $\mathcal{T}$ is productive for a term s, then s rewrites in $\mathcal{T}$ to an infinite constructor term $u_1 : u_2 : u_3 : \ldots$ as its unique infinite normal form.

3 Data-Oblivious Analysis

We formalize the notion of d-o rewriting and introduce the concept of d-o productivity. The idea is a quantitative reasoning where all knowledge about the concrete values of data elements during an evaluation sequence is ignored. For example, consider the following stream specification:

$$\mathsf{M} \to \mathsf{f}(0 : 1 : \mathsf{M}) \quad (1)\ \mathsf{f}(0 : x : \sigma) \to 0 : 1 : \mathsf{f}(\sigma) \quad (2)\ \mathsf{f}(1 : x : \sigma) \to x : \mathsf{f}(\sigma)$$

The specification of M is productive: $\mathsf{M} \to^2 0{:}1{:}\mathsf{f}(\mathsf{M}) \to^3 0{:}1{:}0{:}1{:}\mathsf{f}(\mathsf{f}(\mathsf{M})) \to^* \ldots$. During the rewrite sequence (2) is never applied. Disregarding the identity of data, however, (2) becomes applicable and allows for the rewrite sequence:

$$\mathsf{M} \to \mathsf{f}(\bullet : \bullet : \mathsf{M}) \to^{(2)} \bullet : \mathsf{f}(\mathsf{M}) \to^* \bullet : \mathsf{f}(\bullet : \mathsf{f}(\bullet : \mathsf{f}(\ldots))) ,$$

producing only one element. Hence from the perspective of a data-oblivious analysis there exists a rewrite sequence starting at M that converges to an infinite normal form which has only a stream prefix of length one. In terminology to be introduced in Def. 3.2 we will say that M is not 'd-o productive'.

D-o term rewriting can be thought of as a two-player game between a *rewrite player* $\mathcal{R}$ which performs the usual term rewriting, and an *opponent* $\mathcal{G}$ which before every rewrite step is allowed to arbitrarily exchange data elements for (sort-respecting) data terms in constructor normal form. The opponent can either handicap or support the rewrite player. Respectively, the d-o lower bound on the production of a stream term s is the infimum of the production of s with respect to all possible strategies for the opponent $\mathcal{G}$.

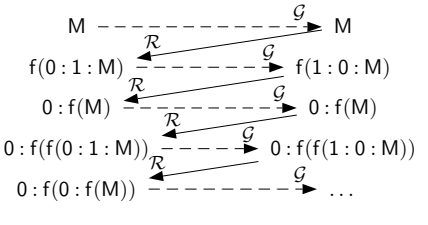


Fig. 3. Data-oblivious rewriting

Fig. 3 depicts d-o rewriting of the above stream specification M; by exchanging data elements, the opponent $\mathcal{G}$ enforces the application of (2). The opponent can be modelled by an operation on stream terms, a function from stream terms to stream terms: $Term(\Sigma)_S \to Term(\Sigma)_S$. For our purposes it will be sufficient to consider strategies for $\mathcal{G}$ with the property that $\mathcal{G}(s)$ is invariant under exchange of data elements in s for all terms s (see Prop. 3.4 below for a formal statement).

Definition 3.1. Let $\mathcal{T} = \langle \Sigma, R \rangle$ be a stream specification. A *data-exchange function on* $\mathcal{T}$ is a function $\mathcal{G} : \mathit{Term}(\Sigma)_S \to \mathit{Term}(\Sigma)_S$ such that $(\!|\mathcal{G}(r)|\!) = (\!|r|\!)$ for all $r \in \mathit{Term}(\Sigma)_S$, and $\mathcal{G}(r)$ is in closed data-constructor normal form.

Definition 3.2. We define the ARS $\mathcal{A}_{\mathcal{T},\mathcal{G}} \subseteq \mathit{Term}(\Sigma)_S \times \mathit{Term}(\Sigma)_S$ for every data-exchange function $\mathcal{G}$, as follows:

$$\mathcal{A}_{\mathcal{T},\mathcal{G}} := \{\langle s, t\rangle \mid s, t \in \mathit{Term}(\Sigma),\ \mathcal{G}(s) \to_{\mathcal{T}} t\} .$$

Thus the steps $s \to_{\mathcal{A}_{\mathcal{T},\mathcal{G}}} t$ in $\mathcal{A}_{\mathcal{T},\mathcal{G}}$ are those of the form $s \mapsto \mathcal{G}(s) \to_{\mathcal{T}} t$.

The *d-o lower bound* $\underline{do}_{\mathcal{T}}(s)$ *on the production of a stream term* $s \in \mathit{Term}(\Sigma)_S$ is defined as follows:

$$\underline{do}_{\mathcal{T}}(s) := \inf\{\Pi_{\mathcal{A}_{\mathcal{T},\mathcal{G}}}(s) \mid \mathcal{G} \text{ a data-exchange function on } \mathcal{T}\} . \qquad (*)$$

A stream specification $\mathcal{T}$ is *d-o productive* if $\underline{do}_{\mathcal{T}}(\mathsf{M}_0) = \infty$ holds.

Proposition 3.3. *For* $\mathcal{T} = \langle \Sigma, R \rangle$ *a stream specification and* $s \in \mathit{Term}(\Sigma)_S$*:*

$$\underline{do}_{\mathcal{T}}(s) \leq \Pi_{\mathcal{T}}(s) .$$

Hence d-o productivity implies productivity.

Proposition 3.4. *The definition of the d-o lower bound* $\underline{do}_{\mathcal{T}}(s)$ *of a stream term* s *in a stream specification* $\mathcal{T}$ *in Def. 3.2 does not change if the quantification in* (*) *is restricted to data-exchange functions* $\mathcal{G}$ *that factor as follows:*

$$\mathcal{G} : \mathit{Term}(\Sigma) \xrightarrow{(\!|\cdot|\!)} \mathit{Term}((\!|\Sigma|\!)) \xrightarrow{\mathcal{G}_\bullet} \mathit{Term}(\Sigma) \qquad \textit{(for some function } \mathcal{G}_\bullet) \qquad (\dagger)$$

(data-exchange functions that are invariant under exchange of data elements).

Proof (Sketch). It suffices to prove that, for every term $s \in \mathit{Term}(\Sigma)_S$, and for every data-exchange function $\mathcal{G}$ on $\mathcal{T}$, there exists a data-exchange function $\mathcal{G}'$ on $\mathcal{T}$ of the form (†) such that $\Pi_{\mathcal{A}_{\mathcal{T},\mathcal{G}'}}(s) \leq \Pi_{\mathcal{A}_{\mathcal{T},\mathcal{G}}}(s)$. This can be shown by adapting $\mathcal{G}$ in an infinite breadth-first traversal over $\mathcal{R}(s)$, the reduction graph of s in $\mathcal{A}_{\mathcal{T},\mathcal{G}}$, thereby defining $\mathcal{G}'$ as follows: if for a currently traversed term s there exists a previously traversed term s_0 with $(\!|s_0|\!) = (\!|s|\!)$ and $\mathcal{G}'(s_0) \neq \mathcal{G}(s)$, then let $\mathcal{G}'(s) := \mathcal{G}'(s_0)$, otherwise let $\mathcal{G}'(s) := \mathcal{G}(s)$. Then the set of terms of the reduction graph $\mathcal{R}'(s)$ of s in $\mathcal{A}_{\mathcal{T},\mathcal{G}'}$ is a subset of the terms in $\mathcal{R}'(s)$. □

Let $\mathcal{T}$ be a stream definition. As an immediate consequence of this proposition we obtain that, for all stream terms $s_1, s_2 \in \mathit{Term}(\Sigma)$ in $\mathcal{T}$, $\underline{do}_{\mathcal{T}}(s_1) = \underline{do}_{\mathcal{T}}(s_2)$ holds whenever $(\!|s_1|\!) = (\!|s_2|\!)$. This fact allows to define d-o lower bounds directly on the data-abstractions of terms: For every term $s \in \mathit{Term}((\!|\Sigma|\!))$, we let $\underline{do}_{\mathcal{T}}((\!|s|\!)) := \underline{do}_{\mathcal{T}}(s)$ for an arbitrarily chosen $s \in \mathit{Term}(\Sigma)_S$. In order to reason about d-o productivity of stream constants (see Sec. 6), we now also introduce lower bounds on the d-o consumption/production behaviour of stream functions.

Definition 3.5. Let $\mathcal{T} = \langle \Sigma, R \rangle$ be a stream specification, $\mathsf{g} \in \Sigma_S^-$, $k = ar_s(\mathsf{g})$, and $\ell = ar_d(\mathsf{g})$. The *d-o lower bound* $\underline{do}_{\mathcal{T}}(\mathsf{g}) : \mathbb{N}^k \to \overline{\mathbb{N}}$ *of* g is:

$$\underline{do}_{\mathcal{T}}(\mathsf{g})(n_1, \ldots, n_k) := \underline{do}_{\mathcal{T}}(\,\mathsf{g}((\bullet^{n_1} : \sigma_1), \ldots, (\bullet^{n_k} : \sigma_k), \underbrace{\bullet, \ldots, \bullet}_{\ell \text{ times}})\,)\,,$$

where $\bullet^m : \sigma := \overbrace{\bullet : \ldots : \bullet}^{m \text{ times}} : \sigma$.

Let $\mathcal{T}$ be a stream specification, and $\mathsf{f} \in \Sigma_{sfun}$ a unary stream function symbol. By a *d-o trace* of f in $\mathcal{T}$ we mean, for a given data-exchange function $\mathcal{G}$, and a closed infinite stream term r of the form $u_0 : u_1 : u_2 : \ldots$, the production function $\pi_\rho : \mathbb{N} \to \overline{\mathbb{N}}$ of a rewrite sequence $\rho\colon\ s_0 = \mathsf{f}(r) \to_{\mathcal{A}_{\mathcal{T},\mathcal{G}}} s_1 \to_{\mathcal{A}_{\mathcal{T},\mathcal{G}}} s_2 \to_{\mathcal{A}_{\mathcal{T},\mathcal{G}}} \ldots$, where π_ρ is defined as follows: for all $n \in \mathbb{N}$, $\pi_\rho(n)$ is the supremum of the lengths of stream prefixes in those terms s_i until which during the steps of ρ less or equal to n stream elements of r within s have been consumed; more precisely, $\pi_\rho(n)$ is the supremum of the number of leading ':' symbols in terms s_i where i is such that no descendent [8, p. 390] of the position of the $(n+1)$-th symbol ':' in s_0 is in the pattern of a redex contracted during the first i steps of ρ.

As a consequence of the use of pattern matching on data in defining rules, even simple stream function specifications can exhibit a complex d-o behaviour, that is, possess large sets of d-o traces. Consider the specification $\mathsf{h}(\mathsf{0} : s) \to \mathsf{h}(s)$ and $\mathsf{h}(\mathsf{1} : s) \to \mathsf{1} : \mathsf{h}(s)$. Here $n \mapsto 0$, and $n \mapsto n$ are d-o traces of h, as well as all functions $h : \mathbb{N} \to \mathbb{N}$ with the property $\forall n \in \mathbb{N}.\ 0 \leq h(n+1) - h(n) \leq 1$. As an example of a more complicated situation, consider the flat function specification:

$$\begin{aligned} \mathsf{f}(\sigma) &\to \mathsf{g}(\sigma, \sigma) \\ \mathsf{g}(\mathsf{0} : y : \sigma, x : \tau) &\to \mathsf{0} : \mathsf{0} : \mathsf{g}(\sigma, \tau) \\ \mathsf{g}(\mathsf{1} : \sigma, x_1 : x_2 : x_3 : x_4 : \tau) &\to \mathsf{0} : \mathsf{0} : \mathsf{0} : \mathsf{0} : \mathsf{0} : \mathsf{g}(\sigma, \tau) \end{aligned}$$

Fig. 4 shows a (small) selection of the set of d-o traces for f, in particular the d-o traces that contribute to the d-o lower bound $\underline{do}_{\mathcal{T}}(\mathsf{f})$. In this example the lower bound $\underline{do}_{\mathcal{T}}(\mathsf{f})$ is a superposition of multiple d-o traces of f. In general $\underline{do}_{\mathcal{T}}(\mathsf{f})$ can even be a superposition of infinitely many d-o traces.

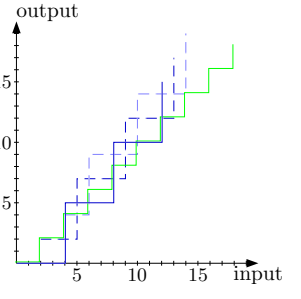


Fig. 4. Traces

4 The Production Calculus

As a means to compute the d-o production behaviour of stream specifications, we introduce a 'production calculus' with periodically increasing functions as its central ingredient.

We use $\overline{\mathbb{N}} := \mathbb{N} \uplus \{\infty\}$, the *extended natural numbers*, with the usual $\leq$, $+$, and we define $\infty - n := \infty$ for all $n \in \mathbb{N}$, and $\infty - \infty := 0$.

An infinite sequence $\sigma \in X^\omega$ is *eventually periodic* if $\sigma = \alpha\beta\beta\beta\ldots$ for some $\alpha \in X^*$ and $\beta \in X^+$. A function $f : \mathbb{N} \to \overline{\mathbb{N}}$ is *eventually periodic* if the sequence $\langle f(0), f(1), f(2), \ldots \rangle$ is eventually periodic.

Definition 4.1. A function $g : \mathbb{N} \to \overline{\mathbb{N}}$ is called *periodically increasing* if it is non-decreasing and the *derivative of* g, $n \mapsto g(n+1) - g(n)$, is eventually periodic. A function $h : \overline{\mathbb{N}} \to \overline{\mathbb{N}}$ is called *periodically increasing* if its restriction to $\mathbb{N}$ is periodically increasing and if $h(\infty) = \lim_{n\to\infty} h(n)$. Finally, a k-ary function $i : (\overline{\mathbb{N}})^k \to \overline{\mathbb{N}}$ is called *periodically increasing* if $i(n_1, ..., n_k) = \min(i_1(n_1), \ldots, i_k(n_k))$ for some unary periodically increasing functions $i_1, \ldots, i_k$.

Periodically increasing (p-i) functions can be denoted by their value at 0 followed by a representation of their derivative. For example, $0\overline{312}$ denotes the p-i function $f : \mathbb{N} \to \mathbb{N}$ with values $0, 3, 4, 6, 7, 9, \ldots$. We use a finer and more flexible notation over the alphabet $\{-, +\}$ that will be useful in Sec. 5. For instance, we denote f as above by the 'io-term' $\langle +^0 - +^3, -+^1 - +^2 \rangle$.

Definition 4.2. An *io-term* is a pair $\langle \alpha, \beta \rangle$ with $\alpha \in \{-, +\}^*$ and $\beta \in \{-, +\}^+$. The set of io-terms is denoted by $\mathcal{I}$, and we use ι, κ to range over io-terms. For $\iota \in \mathcal{I}$, we define $[\![\iota]\!] : \mathbb{N} \to \overline{\mathbb{N}}$, the *interpretation* of $\iota \in \mathcal{I}$, by:

$$[\![\langle -\alpha, \beta \rangle]\!](0) = 0 \qquad [\![\langle +\alpha, \beta \rangle]\!](n) = 1 + [\![\langle \alpha, \beta \rangle]\!](n)$$
$$[\![\langle -\alpha, \beta \rangle]\!](n+1) = [\![\langle \alpha, \beta \rangle]\!](n) \qquad [\![\langle \epsilon, \beta \rangle]\!](n) = [\![\langle \beta, \beta \rangle]\!](n)$$

for all $n \in \mathbb{N}$, and extend it to $\overline{\mathbb{N}} \to \overline{\mathbb{N}}$ by adding $[\![\iota]\!](\infty) = \lim_{n\to\infty} [\![\iota]\!](n)$. We say that ι *represents* $[\![\iota]\!]$. We use $\alpha\overline{\beta}$ as a shorthand for $\langle \alpha, \beta \rangle$. Here ϵ denotes the empty word and we stipulate $[\![\langle \epsilon, +^p \rangle]\!](n) = 1 + 1 + \ldots = \infty$.

It is easy to verify that, for every $\iota \in \mathcal{I}$, the function $[\![\iota]\!]$ is periodically increasing. Furthermore, every p-i function is represented by an io-term. Subsequently, we write $\underline{f}$ for the shortest io-term representing a p-i function $f : \overline{\mathbb{N}} \to \overline{\mathbb{N}}$. Of course we then have $[\![\underline{f}]\!] = f$ for all p-i functions f.

Proposition 4.3. *Unary periodically increasing functions are closed under composition and minimum.*

In addition, these operations can be computed via io-term representations. In [2] we define computable operations $\mathsf{comp} : \mathcal{I} \times \mathcal{I} \to \mathcal{I}$, and $\mathsf{fix} : \mathcal{I} \to \overline{\mathbb{N}}$ such that for all $\iota, \kappa \in \mathcal{I}$: $[\![\mathsf{comp}(\iota, \kappa)]\!] = [\![\iota]\!] \circ [\![\kappa]\!]$ and $\mathsf{fix}(\iota)$ is the least fixed point of $[\![\iota]\!]$.

We introduce a term syntax for the production calculus and rewrite rules for evaluating closed terms; these can be visualized by 'pebbleflow nets', see [3,2].

Definition 4.4. Let $\mathcal{X}$ be a set. The set of *production terms* $\mathcal{P}$ is generated by:

$$p ::= \underline{k} \mid x \mid \underline{f}(p) \mid \mu x.p \mid \mathsf{min}(p, p)$$

where $x \in \mathcal{X}$, for $k \in \overline{\mathbb{N}}$, the symbol $\underline{k}$ is a *numeral* (a term representation) for k, and, for a unary p-i function $f : \overline{\mathbb{N}} \to \overline{\mathbb{N}}$, $\underline{f} \in \mathcal{I}$, the io-term representing f. For every finite set $P = \{p_1, \ldots, p_n\} \subseteq \mathcal{P}$, we use $\mathsf{min}(p_1, \ldots, p_n)$ and $\mathsf{min}\ P$ as shorthands for the production term $\mathsf{min}(p_1, \mathsf{min}(p_2, \ldots, \mathsf{min}(p_{n-1}, p_n)))$.

The *production* $[\![p]\!] \in \overline{\mathbb{N}}$ of a closed production term $p \in \mathcal{P}$ is defined by induction on the term structure, interpreting μ as the least fixed point operator, $\underline{f}$ as f, $\underline{k}$ as k, and min as min.

For faithfully modelling the d-o lower bounds of stream functions with stream arity r, we employ r-ary p-i functions, which we represent by r-ary gates. An *r-ary gate*, abbreviated by $\mathsf{gate}(\iota_1, \ldots, \iota_r)$, is a production term context of the form $\mathsf{min}(\iota_1(\Box_1), \ldots, \iota_r(\Box_r))$, where $\iota_1, \ldots, \iota_r \in \mathcal{I}$. We use γ as a syntactic variable for gates. The *interpretation* of a gate $\gamma = \mathsf{gate}(\iota_1, \ldots, \iota_r)$ is defined as $[\![\gamma]\!](n_1, \ldots, n_r) := \min([\![\iota_1]\!](n_1), \ldots, [\![\iota_r]\!](n_r))$. It is possible to choose unique gate representations $\underline{f}$ of p-i functions f that are efficiently computable from other gate representations, see [2].

Owing to the restriction to (term representations of) periodically increasing functions in Def. 4.4 it is possible to calculate the production $[\![p]\!]$ of terms $p \in \mathcal{P}$. For that purpose, we define a rewrite system which reduces any closed term to a numeral $\underline{k}$. This system makes use of the computable operations comp and fix on io-terms mentioned above.

Definition 4.5. The *rewrite relation* $\to_{\mathsf{R}}$ *on production terms* is defined as the compatible closure of the following rules:

$$\begin{aligned}
\iota_1(\iota_2(p)) &\to \mathsf{comp}(\iota_1, \iota_2)(p) & \qquad \iota(\underline{k}) &\to \underline{[\![\iota]\!](k)} \\
\iota(\mathsf{min}(p_1, p_2)) &\to \mathsf{min}(\iota(p_1), \iota(p_2)) & \qquad \mu x.x &\to \underline{0} \\
\mu x.\mathsf{min}(p_1, p_2) &\to \mathsf{min}(\mu x.p_1, \mu x.p_2) & \qquad \mu x.p &\to p \quad \text{if } x \notin \mathsf{FV}(p) \\
\mu x.\iota(x) &\to \underline{\mathsf{fix}(\iota)} & \qquad \mathsf{min}(\underline{k_1}, \underline{k_2}) &\to \underline{\min(k_1, k_2)}
\end{aligned}$$

The following theorem establishes the usefulness of $\to_{\mathsf{R}}$: the production $[\![p]\!]$ of a production term p can always be computed by reducing p according to $\to_{\mathsf{R}}$, thereby obtaining a normal form that is a numeral after finitely many steps.

Theorem 4.6. *The rewrite relation* $\to_{\mathsf{R}}$ *is confluent, terminating and production preserving, that is,* $p \to_{\mathsf{R}} p'$ *implies* $[\![p]\!] = [\![p']\!]$. *Every closed* $p \in \mathcal{P}$ *has a numeral* $\underline{k}$ *as its unique* $\to_{\mathsf{R}}$*-normal form, and it holds that* $[\![p]\!] = k$.

Proof. Termination of $\to_{\mathsf{R}}$ is straightforward to show. Confluence of $\to_{\mathsf{R}}$ follows by Newman's lemma since all critical pairs are convergent. For preservation of production of $\to_{\mathsf{R}}$ it suffices to show this property for each of the rules. This is not difficult, except for the third rule (that distributes μx over min) for which preservation of production is an immediate consequence of Lem. 4.7 below, in view of the fact that $\langle \overline{\mathbb{N}}, \leq \rangle$ is a complete chain. □

A *complete lattice* is a partially ordered set in which every subset has a least upper bound and a greatest lower bound. A *complete chain* is a complete lattice on which the order is linear. As a consequence of the Knaster–Tarski theorem every order-preserving (non-decreasing) function f on a complete lattice has a least fixed point $\mathrm{lfp}(f)$. We use $\wedge$ for the infix infimum operation.

Lemma 4.7. *Let $\langle D, \leq\rangle$ be a complete chain. Then it holds that:*

$$\forall f, g : D \to D \text{ non-decreasing.} \quad \mathrm{lfp}(f \wedge g) = \mathrm{lfp}(f) \wedge \mathrm{lfp}(g) \qquad (\circ)$$

Proof. Let $\langle D, \leq\rangle$ be a complete chain, and let $f, g : D \to D$ be non-decreasing. The inequality $\mathrm{lfp}(f \wedge g) \leq \mathrm{lfp}(f) \wedge \mathrm{lfp}(g)$ follows easily by using that, for every non-decreasing function h on D, $\mathrm{lfp}(h)$ is the infimum of all pre-fixed points of h, that is, of all $x \in D$ with $h(x) \leq x$. For the converse inequality, let $x := \mathrm{lfp}(f \wedge g)$. Since $x = (f \wedge g)(x) = f(x) \wedge g(x)$, and D is linear, it follows that either $f(x) = x$ or $g(x) = x$, and hence that x is either a fixed point of f or of g. Hence $x \geq \mathrm{lfp}(f)$ or $x \geq \mathrm{lfp}(g)$, and therefore $\mathrm{lfp}(f \wedge g) = x \geq \mathrm{lfp}(f) \wedge \mathrm{lfp}(g)$. □

We additionally mention that ($\circ$) holds in a complete lattice only if it is linear.

5 Translation into Production Terms

In this section we define a translation from stream constants in flat or friendly nesting specifications to production terms. In particular, the root M_0 of a specification $\mathcal{T}$ is mapped by the translation to a production term $[\mathsf{M}_0]$ with the property that if $\mathcal{T}$ is flat (friendly nesting), then the d-o lower bound on the production of M_0 in $\mathcal{T}$ equals (is bounded from below by) the production of $[\mathsf{M}_0]$.

5.1 Translation of Flat and Friendly Nesting Symbols

As a first step of the translation, we describe how for a flat (or friendly nesting) stream function symbol f in a stream specification $\mathcal{T}$ a periodically increasing function $[\mathsf{f}]$ can be calculated that is (that bounds from below) the d-o lower bound on the production of f in $\mathcal{T}$.

Let us again consider the rules (i) $\mathsf{f}(\mathsf{s}(x) : y : \sigma) \to \mathsf{a}(\mathsf{s}(x), y) : \mathsf{f}(y : \sigma)$, and (ii) $\mathsf{f}(\mathsf{0} : \sigma) \to \mathsf{0} : \mathsf{s}(\mathsf{0}) : \mathsf{f}(\sigma)$ from Fig. 2. We model the d-o lower bound on the production of f by a function from $\overline{\mathbb{N}}$ to $\overline{\mathbb{N}}$ defined as the unique solution for X_f of the following system of equations. We disregard what the concrete stream elements are, and therefore we take the infimum over all possible traces:

$$X_\mathsf{f}(n) = \inf\left\{X_{\mathsf{f},(\mathrm{i})}(n), X_{\mathsf{f},(\mathrm{ii})}(n)\right\}$$

where the solutions for $X_{\mathsf{f},(\mathrm{i})}$ and $X_{\mathsf{f},(\mathrm{ii})}$ are the d-o lower bounds of f assuming that the first rule applied in the rewrite sequence is (i) or (ii), respectively. The rule (i) consumes two elements, produces one element and feeds one element back to the recursive call. For rule (ii) these numbers are 1, 2, 0 respectively. Therefore we get:

$$X_{\mathsf{f},(\mathrm{i})}(n) = \text{let } n' := n - 2, \text{ if } n' < 0 \text{ then } 0 \text{ else } 1 + X_\mathsf{f}(n' + 1)\,,$$
$$X_{\mathsf{f},(\mathrm{ii})}(n) = \text{let } n' := n - 1, \text{ if } n' < 0 \text{ then } 0 \text{ else } 2 + X_\mathsf{f}(n' + 0)\,.$$

The unique solution for X_f is $n \mapsto n \dot{-} 1$, represented by the io-term $-\overline{-+}$.

In general, functions may have multiple arguments, which during rewriting may get permuted, duplicated or deleted. The idea is to track single arguments, and take the infimum over all branches in case an argument is duplicated.

For example, the rule $\mathsf{zip}(x:\sigma,\tau) \to x:\mathsf{zip}(\tau,\sigma)$ with a permutation of the stream arguments, gives rise to the following specification:

$$X_{\mathsf{zip},1}(n) = \text{let } n' := n-1, \text{ if } n' < 0 \text{ then } 0 \text{ else } 1 + X_{\mathsf{zip},2}(n')$$
$$X_{\mathsf{zip},2}(n) = \text{let } n' := n-0, \text{ if } n' < 0 \text{ then } 0 \text{ else } 1 + X_{\mathsf{zip},1}(n')\,,$$

and duplication of arguments like in the rule $\mathsf{f}(x:\sigma) \to \mathsf{g}(\sigma, x:\sigma)$ yields:

$$X_{\mathsf{f},1}(n) = \text{let } n' := n-1, \text{ if } n' < 0 \text{ then } 0 \text{ else inf } \left\{X_{\mathsf{g},1}(n'), X_{\mathsf{g},2}(1+n')\right\}.$$

For a recursion variable X let $\langle X\rangle$ be the unique solution for X. The intuition behind the recursion variables is as follows. Let f be a flat stream function symbol with stream arity k. Then the solution $\langle X_\mathsf{f}\rangle$ for X_f models the d-o lower bound on the production of f, that is, $\langle X_\mathsf{f}\rangle = \underline{do}_{\mathcal{T}}(\mathsf{f})$. Furthermore, the variables $X_{\mathsf{f},i}$ for $1 \le i \le k$ describe how the consumption from the i-th argument of f 'retards' the production of f, more precisely, $\langle X_{\mathsf{f},i}\rangle = \lambda n.\underline{do}_{\mathcal{T}}(\mathsf{f}(\bullet^\infty,\ldots,\bullet^\infty,\bullet^n,\bullet^\infty,\ldots,\bullet^\infty))$.

Finally, consider $\mathsf{h}(x:\sigma) \to \mathsf{Y}$, $\mathsf{Y} \to 0:\mathsf{Z}$ and $\mathsf{Z} \to \mathsf{Z}$, a specification illustrating the case of deletion of stream arguments. To translate stream functions like h we extend the translation of flat stream functions to include flat stream constants. To cater for the case that there are no stream arguments or all stream arguments get deleted during reduction, we introduce fresh recursion variables $X_{\mathsf{f},\star}$ for every stream symbol f. The variable $X_{\mathsf{f},\star}$ expresses the production of f assuming infinite supply in each argument, that is, $\langle X_{\mathsf{f},\star}\rangle = \underline{do}_{\mathcal{T}}(\mathsf{f}(\bullet^\infty,\ldots,\bullet^\infty))$.

Therefore in the definition of the translation of stream functions, we need to distinguish the cases according to whether a symbol is weakly guarded or not.

Definition 5.1. We define the *dependency relation* $\multimap$ between symbols in Σ_S^- by $\multimap := \{\langle \mathsf{f},\mathsf{g}\rangle \in \Sigma_S^- \times \Sigma_S^- \mid \mathsf{f}(\boldsymbol{s},\boldsymbol{u}) \to \mathsf{g}(\boldsymbol{t},\boldsymbol{v}) \in R_S\}$ (remember that ':' $\notin \Sigma_S^-$). We say that a symbol $\mathsf{f} \in \Sigma_S^-$ is *weakly guarded* if f is strongly normalising with respect to $\multimap$ and *unguarded*, otherwise.

The translation of a stream function symbol is defined as the unique solution of a (potentially infinite) system of defining equations where the unknowns are functions. More precisely, for each symbol $\mathsf{f} \in \Sigma_{fnest} \supseteq \Sigma_{sfun}$ of a flat or friendly nesting stream specification, this system has a p-i function $[\![\mathsf{f}]\!]$ as a solution for X_f, which is unique among the continuous functions. In [2] we present an algorithm that effectively calculates these solutions in the form of gates.

Definition 5.2. Let $\langle \Sigma, R\rangle$ be a stream specification. For each flat or friendly nesting symbol $\mathsf{f} \in \Sigma_{fnest} \supseteq \Sigma_{flat}$ with arities $k = ar_s(\mathsf{f})$ and $\ell = ar_d(\mathsf{f})$ we

define $[\mathsf{f}] : \overline{\mathbb{N}}^k \to \overline{\mathbb{N}}$, the *translation of* f, as $[\mathsf{f}] := \langle X_\mathsf{f} \rangle$ where $\langle X_\mathsf{f} \rangle$ is the unique solution for X_f of the following system of equations:

For all $n_1, \ldots, n_k \in \overline{\mathbb{N}}$, $i \in \{1, \ldots, k\}$, and $n \in \mathbb{N}$:

$$X_\mathsf{f}(n_1, \ldots, n_k) = \inf \left\{ X_{\mathsf{f},\star}, X_{\mathsf{f},1}(n_1), \ldots, X_{\mathsf{f},k}(n_k) \right\},$$

$$X_{\mathsf{f},\star} = \begin{cases} \inf \left\{ X_{\mathsf{f},\star,\rho} \mid \rho \text{ a defining rule of } \mathsf{f} \right\} & \text{if } \mathsf{f} \text{ is weakly guarded,} \\ 0 & \text{if } \mathsf{f} \text{ is unguarded,} \end{cases}$$

$$X_{\mathsf{f},i}(n) = \begin{cases} \inf \left\{ X_{\mathsf{f},i,\rho}(n) \mid \rho \text{ a defining rule of } \mathsf{f} \right\} & \text{if } \mathsf{f} \text{ is weakly guarded,} \\ 0 & \text{if } \mathsf{f} \text{ is unguarded.} \end{cases}$$

We write $\boldsymbol{u}_i : \sigma_i$ for $u_{i,1} : \ldots : u_{i,p} : \sigma_i$, and $|\boldsymbol{u}_i|$ for p. For $X_{\mathsf{f},\star,\rho}$ and $X_{\mathsf{f},i,\rho}$ we distinguish the possible forms the rule ρ can have. If ρ is nesting, then $X_{\mathsf{f},\star,\rho} = \infty$, and $X_{\mathsf{f},i,\rho}(n) = n$ for all $n \in \overline{\mathbb{N}}$. Otherwise, ρ is non-nesting and of the form:

$$\mathsf{f}((\boldsymbol{u}_1 : \sigma_1), \ldots, (\boldsymbol{u}_k : \sigma_k), v_1, \ldots, v_\ell) \to w_1 : \ldots : w_m : s \,,$$

where either (a) $s \equiv \sigma_j$, or (b) $s \equiv \mathsf{g}((\boldsymbol{u'}_1 : \sigma_{\phi(1)}), \ldots, (\boldsymbol{u'}_{k'} : \sigma_{\phi(k')}), v'_1, \ldots, v'_{\ell'})$ with $k' = ar_s(\mathsf{g})$, $\ell' = ar_d(\mathsf{g})$, and $\phi : \{1, \ldots, k'\} \to \{1, \ldots, k\}$. Then we add:

$$X_{\mathsf{f},\star,\rho} = \begin{cases} \infty & \text{case (a)} \\ m + X_{\mathsf{g},\star} & \text{case (b)} \end{cases}$$

$$X_{\mathsf{f},i,\rho}(n) = \text{let } n' := n - |\boldsymbol{u}_i|, \text{ if } n' < 0 \text{ then } 0 \text{ else}$$
$$m + \begin{cases} n' & \text{case (a), } i = j \\ \infty & \text{case (a), } i \neq j \\ \inf \left\{ X_{\mathsf{g},\star},\ X_{\mathsf{g},j}(n' + |\boldsymbol{u'}_j|) \mid j \in \phi^{-1}(i) \right\} & \text{case (b)} \,. \end{cases}$$

Proposition 5.3. *Let $\mathcal{T}$ be a stream specification, and $\mathsf{f} \in \Sigma_{fnest} \supseteq \Sigma_{flat}$ a stream function symbol with $k = ar_s(\mathsf{f})$. The system of recursive equations described in Def. 5.2 has a k-ary p-i function as its unique solution for X_f, which we denote by $[\mathsf{f}]$. Furthermore, the gate representation $\underline{[\mathsf{f}]}$ of $[\mathsf{f}]$ can be computed.*

Concerning non-nesting rules on which defining rules for friendly nesting symbols depend via $\rightsquigarrow$, this translation uses the fact that their production is bounded below by 'min'. These bounds are not necessarily optimal, but can be used to show productivity of examples like Ex. 2.6.

Example 5.4. Consider a pure stream specification with the function layer:

$$\mathsf{f}(x : \sigma) \to x : \mathsf{g}(\sigma, \sigma, \sigma) \,,$$
$$\mathsf{g}(x : y : \sigma, \tau, \upsilon) \to x : \mathsf{g}(y : \tau, y : \upsilon, y : \sigma) \,.$$

The translation of f is $[\mathsf{f}]$, the unique solution for X_{f} of the system:

$$\begin{aligned}
X_{\mathsf{f}}(n) &= \inf\left\{X_{\mathsf{f},\star},\, X_{\mathsf{f},1}(n)\right\}\\
X_{\mathsf{f},1}(n) &= \text{let } n' := n-1\\
&\qquad \text{if } n'<0 \text{ then } 0 \text{ else } 1+\inf\left\{X_{\mathsf{g},\star},\, X_{\mathsf{g},1}(n'),\, X_{\mathsf{g},2}(n'),\, X_{\mathsf{g},3}(n')\right\}\\
X_{\mathsf{f},\star} &= 1+X_{\mathsf{g},\star}\\
X_{\mathsf{g},1}(n) &= \text{let } n' := n-2,\ \text{if } n'<0 \text{ then } 0 \text{ else } 1+\inf\left\{X_{\mathsf{g},\star},\, X_{\mathsf{g},3}(1+n')\right\}\\
X_{\mathsf{g},2}(n) &= 1+\inf\left\{X_{\mathsf{g},\star},\, X_{\mathsf{g},1}(1+n)\right\}\\
X_{\mathsf{g},3}(n) &= 1+\inf\left\{X_{\mathsf{g},\star},\, X_{\mathsf{g},2}(1+n)\right\}\\
X_{\mathsf{g},\star} &= 1+X_{\mathsf{f},\star}
\end{aligned}$$

An algorithm for solving such systems of equations is described in [2]; here we solve the system directly. Note that $X_{\mathsf{f},\star} = X_{\mathsf{g},\star} = \infty$, and therefore $X_{\mathsf{g},3}(n) = 1 + X_{\mathsf{g},2}(n+1) = 2 + X_{\mathsf{g},1}(n+2) = 3 + X_{\mathsf{g},3}(n)$, hence $\forall n \in \mathbb{N}.\ X_{\mathsf{g},3}(n) = \infty$. Likewise we obtain $X_{\mathsf{g},2}(n) = \infty$ if $n \geq 1$ and 1 for $n = 0$, and $X_{\mathsf{g},1}(n) = \infty$ if $n \geq 2$ and 0 for $n \leq 1$. Then if follows that $[\mathsf{f}](0) = 0$, $[\mathsf{f}](1) = [\mathsf{f}](2) = 1$, and $[\mathsf{f}](n) = \infty$ for all $n \geq 2$, represented by the gate $\underline{[\mathsf{f}]} = \mathsf{gate}(-+\overline{--+})$. The gate corresponding to g is $\underline{[\mathsf{g}]} = \mathsf{gate}(\overline{--+}, \overline{+-+}, \overline{+})$.

Example 5.5. Consider a flat stream function specification with the following rules which use pattern matching on the data constructors $\mathsf{0}$ and $\mathsf{1}$:

$$\mathsf{f}(\mathsf{0}:\sigma) \to \mathsf{g}(\sigma) \qquad \mathsf{f}(\mathsf{1}:x:\sigma) \to x:\mathsf{g}(\sigma) \qquad \mathsf{g}(x:y:\sigma) \to x:y:\mathsf{g}(\sigma)$$

denoted ρ_{f_0}, ρ_{f_1}, and ρ_{g}, respectively. Then, $[\mathsf{f}]$ is the solution for $X_{\mathsf{f},1}$ of:

$$\begin{aligned}
X_{\mathsf{f}}(n) &= \inf\left\{X_{\mathsf{f},\star},\, X_{\mathsf{f},1}(n)\right\}\\
X_{\mathsf{f},1}(n) &= \inf\left\{X_{\mathsf{f},1,\rho_{\mathsf{f}_0}}(n),\, X_{\mathsf{f},1,\rho_{\mathsf{f}_1}}(n)\right\}\\
X_{\mathsf{f},1,\rho_{\mathsf{f}_0}}(n) &= \text{let } n' := n-1,\ \text{if } n'<0 \text{ then } 0 \text{ else } \left\{X_{\mathsf{g},\star},\, X_{\mathsf{g},1}(n')\right\}\\
X_{\mathsf{f},1,\rho_{\mathsf{f}_1}}(n) &= \text{let } n' := n-2,\ \text{if } n'<0 \text{ then } 0 \text{ else } 1+\left\{X_{\mathsf{g},\star},\, X_{\mathsf{g},1}(n')\right\}\\
X_{\mathsf{f},\star} &= \inf\left\{X_{\mathsf{g},\star},\, 1+X_{\mathsf{g},\star}\right\}\\
X_{\mathsf{g},1}(n) &= \text{let } n' := n-2,\ \text{if } n'<0 \text{ then } 0 \text{ else } 2+\left\{X_{\mathsf{g},\star},\, X_{\mathsf{g},1}(n')\right\}\\
X_{\mathsf{g},\star} &= 2+X_{\mathsf{g},\star}\,.
\end{aligned}$$

As solution we obtain an overlapping of both traces $\underline{[\mathsf{f}]}_{1,\rho_{\mathsf{f}_0}}$ and $[\mathsf{f}]_{1,\rho_{\mathsf{f}_1}}$, that is, $[\mathsf{f}]_1(n) = n \dot{-} 2$ represented by the gate $\underline{[\mathsf{f}]} = \mathsf{gate}(--\overline{-+})$.

The following lemma states that the translation $[\mathsf{f}]$ of a flat stream function symbol f (as defined in Def. 5.2) is the d-o lower bound on the production function of f. For friendly nesting stream symbols f it states that $[\mathsf{f}]$ pointwisely bounds from below the d-o lower bound on the production function of f.

Lemma 5.6. *Let $\mathcal{T}$ be a stream specification, and let $\mathsf{f} \in \Sigma_{fnest} \supseteq \Sigma_{flat}$.*

(i) If f is flat, then: $[\mathsf{f}] = \underline{do}_{\mathcal{T}}(\mathsf{f})$. Hence, $\underline{do}_{\mathcal{T}}(\mathsf{f})$ is periodically increasing.
(ii) If f is friendly nesting, then it holds: $[\mathsf{f}] \leq \underline{do}_{\mathcal{T}}(\mathsf{f})$ (pointwise inequality).

5.2 Translation of Stream Constants

In the second step, we now define a translation of stream constants in a flat or friendly nesting stream specification into production terms under the assumption that gate translations for the stream functions are given. Here the idea is that the recursive definition of a stream constant M is unfolded step by step; the terms thus arising are translated according to their structure using gate translations of the stream function symbols from a given family of gates; whenever a stream constant is met that has been unfolded before, the translation stops after establishing a binding to a μ-binder created earlier.

Definition 5.7. Let $\mathcal{T}$ be a stream specification, $\mathsf{M} \in \Sigma_{scon}$, and $\mathcal{F} = \{\gamma_\mathsf{f}\}_{\mathsf{f}\in\Sigma_{sfun}}$ a family of gates. The *translation* $[\mathsf{M}]^\mathcal{F} \in \mathcal{P}$ *of* M *with respect to* $\mathcal{F}$ is defined by $[\mathsf{M}]^\mathcal{F} := [\mathsf{M}]^\mathcal{F}_\varnothing$, where, for every $\mathsf{M} \in \Sigma_{scon}$ and every $\alpha \subseteq \Sigma_{scon}$ we define:

$$[\mathsf{M}(\boldsymbol{u})]^\mathcal{F}_\alpha := [\mathsf{M}]^\mathcal{F}_\alpha := \begin{cases} \mu M.\mathsf{min}\,\{[r]^\mathcal{F}_{\alpha\cup\{\mathsf{M}\}} \mid \mathsf{M}(\boldsymbol{v}) \to r \in R\} & \text{if } \mathsf{M} \notin \alpha \\ M & \text{if } \mathsf{M} \in \alpha \end{cases}$$

$$[u : s]^\mathcal{F}_\alpha := +\overline{-+}([s]^\mathcal{F}_\alpha)$$

$$[\mathsf{f}(s_1,\ldots,s_{ar_\mathsf{s}(\mathsf{f})},u_1,\ldots,u_{ar_\mathsf{d}(\mathsf{f})})]^\mathcal{F}_\alpha := \gamma_\mathsf{f}([s_1]^\mathcal{F}_\alpha,\ldots,[s_{ar_\mathsf{s}(\mathsf{f})}]^\mathcal{F}_\alpha)$$

Example 5.8. As an example we translate Pascal's triangle, see Fig. 2. The translation of the stream function symbols is $\mathcal{F} = \{\underline{[\mathsf{f}]} = \mathsf{gate}(-\overline{-+})\}$, cf. page 90. Hence we obtain $[\mathsf{P}]^\mathcal{F} = \mu P.+\overline{-+}(+\overline{-+}(-\overline{-+}(P)))$ as the translation of P.

The following lemma is the basis of our main results in Sec. 6. It entails that if we use gates that represent d-o optimal lower bounds on the production of the stream functions, then the translation of a stream constant M yields a production term that rewrites to the d-o lower bound of the production of M.

Lemma 5.9. *Let* $\mathcal{T} = \langle\Sigma, R\rangle$ *be a stream specification, and* $\mathcal{F} = \{\gamma_\mathsf{f}\}_{\mathsf{f}\in\Sigma_{sfun}}$ *a family of gates. If* $[\![\gamma_\mathsf{f}]\!] = \underline{do}_\mathcal{T}(\mathsf{f})$ *for all* $\mathsf{f} \in \Sigma_{sfun}$*, then for all* $\mathsf{M} \in \Sigma_{scon}$*:* $[\![[\mathsf{M}]^\mathcal{F}]\!] = \underline{do}_\mathcal{T}(\mathsf{M})$*. Hence,* $\mathcal{T}$ *is d-o productive if and only if* $[\![[\mathsf{M}_0]^\mathcal{F}]\!] = \infty$*.*

If $[\![\gamma_\mathsf{f}]\!] \le \underline{do}_\mathcal{T}(\mathsf{f})$ *for all* $\mathsf{f} \in \Sigma_{sfun}$*, then for all* $\mathsf{M} \in \Sigma_{scon}$*:* $[\![[\mathsf{M}]^\mathcal{F}]\!] \le \underline{do}_\mathcal{T}(\mathsf{M})$*. Consequently,* $\mathcal{T}$ *is d-o productive if* $[\![[\mathsf{M}_0]^\mathcal{F}]\!] = \infty$*.*

6 Deciding Data-Oblivious Productivity

In this section we assemble our results concerning decision of d-o productivity, and automatable recognition of productivity. We define methods:

- (DOP) for deciding d-o productivity of flat stream specifications,
- (DP) for deciding productivity of pure stream specifications, and
- (RP) for recognising productivity of friendly nesting stream specifications,

that proceed in the following steps:

(i) Take as input a (DOP) flat, (DP) pure, or (RP) friendly nesting stream specification $\mathcal{T} = \langle\Sigma, R\rangle$.

(ii) Translate the stream function symbols into gates $\mathcal{F} := \{\underline{[\mathsf{f}]}\}_{\mathsf{f}\in\Sigma_{sfun}}$ (Def. 5.2).
(iii) Construct the production term $[\mathsf{M}_0]^{\mathcal{F}}$ with respect to $\mathcal{F}$ (Def. 5.7).
(iv) Compute the production k of $[\mathsf{M}_0]^{\mathcal{F}}$ using $\to_{\mathsf{R}}$ (Def. 4.5).
(v) Give the following output:
(DOP) "$\mathcal{T}$ is d-o productive" if $k = \infty$, else "$\mathcal{T}$ is not d-o productive".
(DP) "$\mathcal{T}$ is productive" if $k = \infty$, else "$\mathcal{T}$ is not productive".
(RP) "$\mathcal{T}$ is productive" if $k = \infty$, else "don't know".

Note that all of these steps are automatable (cf. our productivity tool, Sec. 7).

Our main result states that d-o productivity is decidable for flat stream specifications. Since d-o productivity implies productivity (Prop. 3.3), we obtain a computable, d-o optimal, sufficient condition for productivity of flat stream specifications, which cannot be improved by any other d-o analysis. Second, since for pure stream specifications d-o productivity and productivity are the same, we get that productivity is decidable for them.

Theorem 6.1. *(i)* DOP *decides d-o productivity of flat stream specifications, (ii)* DP *decides productivity of pure stream specifications.*

Proof. Let k be the production of the term $[\mathsf{M}_0]^{\mathcal{F}} \in \mathcal{P}$ in step (iv) of DOP/DP.
(i) By Lem. 5.6 (i), Lem. 5.9, and Thm. 4.6 we find: $k = \underline{do}_{\mathcal{T}}(\mathsf{M}_0)$.
(ii) For pure specifications we additionally note: $\Pi_{\mathcal{T}}(\mathsf{M}_0) = \underline{do}_{\mathcal{T}}(\mathsf{M}_0)$. □

Third, we obtain a computable, sufficient condition for productivity of friendly nesting stream specifications.

Theorem 6.2. *A friendly nesting (flat) stream specification $\mathcal{T}$ is productive if the algorithm* RP*(*DOP*) recognizes $\mathcal{T}$ as productive.*

Proof. By Lem. 5.6 (ii), Lem. 5.9, and Thm. 4.6: $k \leq \underline{do}_{\mathcal{T}}(\mathsf{M}_0) \leq \Pi_{\mathcal{T}}(\mathsf{M}_0)$. □

Example 6.3. We illustrate the decision of d-o productivity by means of Pascal's triangle, Fig. 2. We reduce $[\mathsf{P}]^{\mathcal{F}}$, the translation of P, to $\to_{\mathsf{R}}$-normal form:

$$[\mathsf{P}]^{\mathcal{F}} = \mu P.+\overline{-+}(+\overline{-+}(-\overline{-+}(P))) \to_{\mathsf{R}}^{*} \mu P.++-\overline{-+}(P) \to_{\mathsf{R}} \underline{\infty}$$

Hence $\underline{do}_{\mathcal{T}}(\mathsf{P}) = \infty$, and P is d-o productive and therefore productive.

7 Conclusion and Further Work

In order to formalize quantitative approaches for recognizing productivity of stream specifications, we defined the notion of d-o rewriting and investigated d-o productivity. For the syntactic class of flat stream specifications (that employ pattern matching on data), we devised a decision algorithm for d-o productivity. In this way we settled the productivity recognition problem for flat stream specifications from a d-o perspective. For the even larger class including friendly nesting stream function rules, we obtained a computable sufficient condition for productivity. For the subclass of pure stream specifications (a substantial extension of the class given in [3]) we showed that productivity and d-o productivity

coincide, and thereby obtained a decision algorithm for productivity of pure specifications.

We have implemented in Haskell the decision algorithm for d-o productivity. This tool, together with more information including a manual, examples, our related papers, and a comparison of our criteria with those of [4,7,1] can be found at our web page `http://infinity.few.vu.nl/productivity`. The reader is invited to experiment with our tool.

It is not possible to obtain a d-o optimal criterion for non-productivity of flat specifications in an analogous way to how we established such a criterion for productivity. This is because the d-o upper bound $\overline{do}_{\mathcal{T}}(\mathsf{f})$ on the production of a stream function f in flat stream specifications is not in general a periodically increasing function. For example, for the following stream function specification:

$$\mathsf{f}(x:\sigma,\tau) \to x:\mathsf{f}(\sigma,\tau)\;, \qquad\qquad \mathsf{f}(\sigma,y:\tau) \to y:\mathsf{f}(\sigma,\tau)\;,$$

it holds that $\overline{do}(\mathsf{f})(n_1,n_2) = n_1+n_2$, which is not p-i. While this example is not orthogonal, $\overline{do}(\mathsf{f})$ is also not p-i for the following similar orthogonal example:

$$\mathsf{f}(\mathsf{0}:x:\sigma,y:\tau) \to x:\mathsf{f}(\sigma,\tau)\;, \qquad\qquad \mathsf{f}(\mathsf{1}:\sigma,x:y:\tau) \to y:\mathsf{f}(\sigma,\tau)\;.$$

Currently we are developing a method that goes beyond a d-o analysis, one that would, e.g., prove productivity of the example B given in the introduction. Moreover, we study a refined production calculus that accounts for the delay of evaluation of stream elements, in order to obtain a faithful modelling of lazy evaluation, needed for example for S on page 82, where the first element depends on a 'future' expansion of S.

Acknowledgement. We thank Jan Willem Klop, Carlos Lombardi, Vincent van Oostrom, and Roel de Vrijer for useful discussions, and the anonymous referees for their comments and suggestions.

References

1. Buchholz, W.: A Term Calculus for (Co-)Recursive Definitions on Streamlike Data Structures. Annals of Pure and Applied Logic 136(1-2), 75–90 (2005)
2. Endrullis, J., Grabmayer, C., Hendriks, D.: Data-Oblivious Stream Productivity. Technical report (2008), `http://arxiv.org/pdf/0806.2680`
3. Endrullis, J., Grabmayer, C., Hendriks, D., Isihara, A., Klop, J.W.: Productivity of Stream Definitions. In: Csuhaj-Varjú, E., Ésik, Z. (eds.) FCT 2007. LNCS, vol. 4639, pp. 274–287. Springer, Heidelberg (2007)
4. Hughes, J., Pareto, L., Sabry, A.: Proving the Correctness of Reactive Systems Using Sized Types. In: POPL 1996, pp. 410–423 (1996)
5. Kapur, D., Narendran, P., Rosenkrantz, D.J., Zhang, H.: Sufficient-Completeness, Ground-Reducibility and their Complexity. Acta Informatica 28(4), 311–350 (1991)
6. Sijtsma, B.A.: On the Productivity of Recursive List Definitions. ACM Transactions on Programming Languages and Systems 11(4), 633–649 (1989)
7. Telford, A., Turner, D.: Ensuring Streams Flow. In: AMAST, pp. 509–523 (1997)
8. Terese: Term Rewriting Systems. Cambridge Tracts in Theoretical Computer Science, vol. 55. Cambridge University Press, Cambridge (2003)

Reasoning about XML with Temporal Logics and Automata

Leonid Libkin[1] and Cristina Sirangelo[1,2]

[1] University of Edinburgh
[2] LSV, ENS–Cachan, INRIA

Abstract. We show that problems arising in static analysis of XML specifications and transformations can be dealt with using techniques similar to those developed for static analysis of programs. Many properties of interest in the XML context are related to navigation, and can be formulated in temporal logics for trees. We choose a logic that admits a simple single-exponential translation into unranked tree automata, in the spirit of the classical LTL-to-Buchi automata translation. Automata arising from this translation have a number of additional properties; in particular, they are convenient for reasoning about unary node-selecting queries, which are important in the XML context. We give two applications of such reasoning: one deals with a classical XML problem of reasoning about navigation in the presence of schemas, and the other relates to verifying security properties of XML views.

Keywords: Query automata, static analysis, temporal logics, XML.

1 Introduction

Static analysis of XML specifications and transformations has been the focus of many recent papers (see, e.g., [1,4,6,8,10,13,16,24,25,35]). Typical examples include consistency of type declarations and constraints, or of schema specifications and navigational properties, or containment of XPath expressions. They found application in query optimization, access control, data exchange, and reasoning about security properties of views, among others.

Many XML specifications – for example, various schema formalisms – are automata-based. Furthermore, there is a close connection between XML navigation, which is a key component of query languages, and temporal logics used in the field of verification [5,26,25,22,16]. Thus, it is very natural to adapt automata-based techniques developed by the verification community (cf. [11]) for XML static analysis problems involving schemas and navigation.

Examples of such usage exist, but by and large they take existing verification tools, and attempt to reshape the problem at hand so that those tools would be applicable to it. For example, [25] shows how to reason about XML navigation language XPath and XML schemas by encoding them in PDL. While it achieves provably optimal EXPTIME-bound, it does so by a rather complicated algorithm (for example, it uses, as a black box for one of its steps, a translation

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 97–112, 2008.

from PDL into a certain type of tree automata [40], for which no efficient implementations exist). Another example of such reasoning [16] goes via much better implementable μ-calculus, but the technique only guarantees $n^{O(n)}$ algorithms for problems for which $2^{O(n)}$ algorithms exist.

We propose an alternative approach: instead of using verification techniques as-is in the XML context, we adapt them to get better static analysis algorithms. The present paper can be viewed as a proof-of-concept paper: we demonstrate one logic-to-automata translation targeted to XML applications, which closely resembles the standard Vardi-Wolper's LTL-to-Büchi translation [39], and show that it is easily applicable in two typical XML reasoning tasks.

Typically, temporal logic formulae are translated into either nondeterministic or alternating automata; for LTL, both are possible [39,37]. We believe that both should be explored in the XML context. For this paper, we concentrate on the former. A recent workshop paper [10] developed an alternating-automata based approach; it handled a more expressive navigation language, but did not work out connections with XML schemas, as we do here.

Our goal is to find a clean direct translation from a logical formalism suitable for expressing many XML reasoning tasks, into an automata model. Towards that end, we use a simple LTL-like logic for trees, which we call $\mathrm{TL}^{\mathrm{tree}}$, rather than a W3C-designed language (but we shall show that such languages can be easily translated into $\mathrm{TL}^{\mathrm{tree}}$). This logic was defined in [34], and it was recently used in the work on XPath extensions [26], and as a key ingredient for an expressively-complete logic for reasoning about procedural programs [2,3].

The translation will produce a bit more than automata rejecting or accepting trees; instead it will produce *query automata* [30,28,15] which can also select nodes from trees in their successful runs. The ability to produce such automata is not surprising at all (since in the Vardi-Wolper construction states are sets of formulae and successful runs tell us which formulae hold in which positions). Nonetheless, it is a very useful feature for XML reasoning, since many XML data processing tasks are about *node-selecting queries* [18,30,36,29]. Furthermore, additional properties of query automata arising in the translation make operations such as complementation and testing containment very easy for them. Consequently, it becomes easier to combine several reasoning tasks.

Organization In Section 2 we give examples of XML reasoning where the logic/automata connection would be useful. Section 3 describes unranked trees and automata for them. In Section 4 we present the logic $\mathrm{TL}^{\mathrm{tree}}$ and various XPath formalisms, and give an easy translation of XPath into $\mathrm{TL}^{\mathrm{tree}}$. In Section 5 we give a translation from $\mathrm{TL}^{\mathrm{tree}}$ to query automata. Section 6 applies this translation in complex reasoning tasks involving schemas and navigation in XML documents, and Section 7 gives an application to reasoning about XML views.

2 Motivating Examples

We now consider two examples of XML static analysis problems that will later be handled by restating these problems with the help of $\mathrm{TL}^{\mathrm{tree}}$ and the automata

translation. While formal definitions will be given later, for the reader not fluent in XML the following abstractions will be sufficient. First, XML documents themselves are labeled unranked trees (that is, different nodes can have a different number of children). XML schemas describe how documents are structured; they may be given by several formalisms that are all subsumed by tree automata. The most common of such formalisms is referred to as DTDs (document type definitions). And finally XPath is a navigational language; an XPath expression for now can be thought of as selecting a set of nodes in a tree.

Reasoning about schemas and navigation. A common static analysis problem in the XML context, arising in query optimization and consistency checking, is the interaction of navigational properties (expressed, for example, in XPath) with schemas (often given as DTDs). Known results about the complexity of problems such as XPath containment [35], or XPath/DTD consistency [6], are typically stated in terms of completeness for various intractable complexity classes. They imply unavoidability of exponential-time algorithms, but they do not necessarily lead to reasonable algorithms that can be used in practice.

To illustrate this, consider the *containment problem* of XPath expressions under a DTD, i.e., checking whether for all trees satisfying a DTD d, the set of nodes selected by e_1 is contained in the set selected by e_2 (written as $d \models e_1 \subseteq e_2$). Automata-based algorithms would either translate XPath directly into automata (which could depend heavily on a particular syntactic class [31]), or attempt a generic translation via an existing logic. The second approach, taken by [25,6,16], translates e_1, e_2, and d into formulae of expressive logics such as PDL (in [25]) or μ-calculus (in [16]). Then one uses techniques of [40,38] to check if there exists a finite tree T satisfying d and a node s in T witnessing $e_1(s) \wedge \neg e_2(s)$, i.e., a counterexample to the containment. PDL and μ-calculus have been chosen because of their ability to encode XML schemas, e.g., DTDs, but, as we shall see, this is easy to avoid.

While this is very much in the spirit of the traditional logic/automata connection used so extensively in static analysis of programs, there are some problems with this approach as currently used. The logics used were chosen because of their ability to encode DTDs, but this makes the constructions apply several algorithms as black-boxes. For example, the PDL construction of [25] combines a translation into PDL with converse on binary trees, a rather complex automata model of [40] together with an extra automaton that restricts it to finite trees. Second, we do not get a concise description of the set of all possible counterexamples, rather a yes or no answer. And third, the high expressiveness of logics comes at a cost. The running time of algorithms that go via μ-calculus is $n^{O(n)}$ [16]. For the PDL approach [25], the running time is $2^{O(\|e_1\|+\|e_2\|+\|d\|)}$, where $\|\cdot\|$ denotes the size. In several applications, we would rather avoid the $2^{O(\|d\|)}$ factor, since many DTDs are computer-generated from database schemas and could be very large, while XPath expressions tend to be small.

The translation we propose is a direct and simple construction, and does not rely on complicated algorithms such as the PDL-to-automata translation. It produces a concise description of *all* possible counterexamples, which can be

reused later. Finally, it exhibits an exponential blowup in the size of e_1 and e_2, but remains polynomial in the size of the schema.

Reasoning about views and query answers. Often the user sees not a whole XML document, but just a portion of it, V (called a *view*), generated by a query. Such a query typically specifies a set of nodes selected from a source document, and thus can be represented by a query automaton $\mathcal{QA}_{\mathcal{V}}$: i.e., an extension of a tree automaton that can select nodes in trees; a formal definition will be given shortly.

If we only have access to V, we do not know the source document that produced it, as there could be many trees T satisfying $V = \mathcal{QA}_{\mathcal{V}}(T)$. We may know, however, that every such source has to satisfy some schema requirements, presented by a tree automaton $\mathcal{A}$. A common problem is to check whether V may reveal some information about the source. If $\mathfrak{Q}$ is a Boolean (yes/no) query, one defines the *certain answer* to $\mathfrak{Q}$ over V to be true iff $\mathfrak{Q}$ is true in every possible T that generates V:

$$\underline{certain}^{\mathcal{A}}_{\mathcal{QA}_{\mathcal{V}}}(\mathfrak{Q}; V) \quad = \quad \bigwedge\{\mathfrak{Q}(T) \;\mid\; V = \mathcal{QA}_{\mathcal{V}}(T), \;\; T \text{ is accepted by } \mathcal{A}\}$$

Now if by looking at V, we can conclude that $\underline{certain}^{\mathcal{A}}_{\mathcal{QA}_{\mathcal{V}}}(\mathfrak{Q}; V)$ is true, then V reveals that $\mathfrak{Q}$ is true in an unknown source. If $\mathfrak{Q}$ is a containment statement $e_1 \subseteq e_2$, such an inclusion could be information that needs to be kept secret (e.g., it may relate two different groups of people). For more on this type of applications, see [13,14].

Suppose $\mathfrak{Q}$ itself is definable by an automaton $\mathcal{A}_{\mathfrak{Q}}$. If we can convert automata $\mathcal{A}_{\mathfrak{Q}}$, $\mathcal{A}$, and the query automaton $\mathcal{QA}_{\mathcal{V}}$ into a new automaton $\mathcal{A}^*$ that accepts V iff $\underline{certain}^{\mathcal{A}}_{\mathcal{QA}_{\mathcal{V}}}(\mathfrak{Q}; V)$ is false, then acceptance by $\mathcal{A}^*$ gives us some assurances that the secret is not revealed. Furthermore, since views are often given by XPath expressions, and e_1 and e_2 are often XPath expressions too, an efficient algorithm for constructing $\mathcal{A}^*$ would give us a verification algorithm exponential in (typically short) XPath expressions defining e_1, e_2, and $\mathcal{V}$, and polynomial in a (potentially large) expression defining the schema.

In fact, we shall present a polynomial-time construction for $\mathcal{A}^*$ for the case of subtree- (or upward-closed) queries [7]. In that case, combining it with previous efficient translations from logical formulae into query automata, we get efficient algorithms for verifying properties of views.

3 Unranked Trees and Automata

Unranked trees. XML documents are normally abstracted as labeled unranked trees. We now recall the standard definitions, see [29,22,36]. Nodes in unranked trees are elements of $\mathbb{N}^*$, i.e. strings of natural numbers. We write $s \cdot s'$ for the concatenation of strings, and ε for the empty string. The basic binary relations on $\mathbb{N}^*$ are the *child relation*: $s \prec_{\text{ch}} s'$ if $s' = s \cdot i$, for some $i \in \mathbb{N}$, and the *next-sibling* relation: $s' \prec_{\text{ns}} s''$ if $s' = s \cdot i$ and $s'' = s \cdot (i+1)$ for some $s \in \mathbb{N}^*$

and $i \in \mathbb{N}$. The *descendant* relation $\prec_{\text{ch}}^*$ and the younger sibling relation $\prec_{\text{ns}}^*$ are the reflexive-transitive closures of $\prec_{\text{ch}}$ and $\prec_{\text{ns}}$.

An *unranked tree domain* D is a finite prefix-closed subset of $\mathbb{N}^*$ such that $s \cdot i \in D$ implies $s \cdot j \in D$ for all $j < i$. If Σ is a finite alphabet, an *unranked tree* is a pair $T = (D, \lambda)$, where D is a tree domain and λ is a labeling function $\lambda : D \to \Sigma$.

Unranked tree automata and XML schemas. A *nondeterministic unranked tree automaton* (cf. [29,36]) over Σ-labeled trees is a triple $\mathcal{A} = (Q, F, \delta)$ where Q is a finite set of states, $F \subseteq Q$ is the set of final states, and δ is a mapping $Q \times \Sigma \to 2^{Q^*}$ such that each $\delta(q, a)$ is a regular language over Q. We assume that each $\delta(q, a)$ is given as an NFA. A *run* of $\mathcal{A}$ on a tree $T = (D, \lambda)$ is a function $\rho_{\mathcal{A}} : D \to Q$ such that if $s \in D$ is a node with n children, and $\lambda(s) = a$, then the string $\rho_{\mathcal{A}}(s \cdot 0) \cdots \rho_{\mathcal{A}}(s \cdot (n-1))$ is in $\delta(\rho_{\mathcal{A}}(s), a)$. Thus, if s is a leaf labeled a, then $\rho_{\mathcal{A}}(s) = q$ implies that $\varepsilon \in \delta(q, a)$. A run is *accepting* if $\rho_{\mathcal{A}}(\varepsilon) \in F$, and a tree is *accepted* by $\mathcal{A}$ if an accepting run exists. Sets of trees accepted by automata $\mathcal{A}$ are called regular and denoted by $L(\mathcal{A})$.

There are multiple notions of *schemas* for XML documents, DTDs being the most popular one. What is common for them is that they are subsumed by the power of unranked tree automata, and each specific formalism has a simple (often linear time) translation into an automaton [36]. So when we speak of XML schemas, we shall assume that they are given by unranked tree automata.

Query automata. It is well known that automata capture the expressiveness of MSO sentences over finite and infinite strings and trees. The model of query automata [30] captures the expressiveness of MSO formulae $\varphi(x)$ with one free first-order variable – that is, MSO-definable unary queries. We present here a nondeterministic version, as in [28,15].

A *query automaton (QA)* for Σ-labeled unranked trees is a tuple $\mathcal{QA} = (Q, F, Q_s, \delta)$, where (Q, F, δ) is an unranked tree automaton, and $Q_s \subseteq Q$ is the set of *selecting states*. Each run ρ of $\mathcal{QA}$ on a tree $T = (D, \lambda)$ defines the set $S_\rho(T) = \{s \in D \mid \rho(s) \in Q_s\}$ of nodes assigned a selecting state. The unary query defined by $\mathcal{QA}$ is then, under the *existential semantics*,

$$\mathcal{QA}^{\exists}(T) \;=\; \bigcup \{S_\rho(T) \mid \rho \text{ is an accepting run of } \mathcal{QA} \text{ on } T\}.$$

Dually, one can define $\mathcal{QA}^{\forall}(T)$ under the *universal semantics* as the intersection of $S_\rho(T)$'s. Both semantics capture the class of unary MSO queries [28].

These notions are not very convenient for reasoning tasks, as many runs need to be taken into account – different nodes may be selected in different runs. Also, it makes operations on query automata hard computationally: for example, a natural notion of complement for an existential-semantics QA will be expressed as a universal semantics QA, requiring an exponential time algorithm to convert it back into an existential QA.

To remedy this, we define a notion of *single-run query automata* as QAs (Q, F, Q_s, δ) satisfying two conditions:

1. For every tree T, and accepting runs ρ_1 and ρ_2, we have $S_{\rho_1}(T) = S_{\rho_2}(T)$; and
2. The automaton (Q, F, δ) accepts every tree.

For such QAs, we can unambiguously define the set of selected nodes as $\mathcal{QA}(T) = S_\rho(T)$, where ρ is an arbitrarily chosen accepting run.

While the conditions are fairly strong, they do not restrict the power of QAs:

Fact 1. (see [15,32,33]) *For every query automaton $\mathcal{QA}$, there exists an equivalent single-run query automaton, that is, a single-run query automaton $\mathcal{QA}'$ such that $\mathcal{QA}^{\exists}(T) = \mathcal{QA}'(T)$ for every tree T.*

Remarks: the construction in [15] needs a slight modification to produce such QA; also it needs to be extended to unranked trees which is straightforward. This was also noticed in [33]. One can also get this result by slightly adapting the construction of [32].

We now make a few remarks about closure properties and decision problems for single-run QAs. It is known [29] that nonemptiness problem for existential-semantics QAs is solvable in polynomial time; hence the same is true for single-run QAs. Single-run QAs are easily closed under intersection: the usual product construction works. Moreover, if one takes a product $\mathcal{A} \times \mathcal{QA}$ of a tree automaton and a single-run QA (where selecting states are pairs containing a selecting state of $\mathcal{QA}$), the result is a QA satisfying 1) above, and the nonemptiness problem for it is solvable in polynomial time too.

We define the *complement* of a single-run QA as $\overline{\mathcal{QA}} = (Q, F, Q - Q_s, \delta)$, where $\mathcal{QA} = (Q, F, Q_s, \delta)$. It follows immediately from the definition that for every tree T with domain D, we have $\overline{\mathcal{QA}}(T) = D - \mathcal{QA}(T)$, if $\mathcal{QA}$ is single-run. This implies that the *containment problem* $\mathcal{QA}_1 \subseteq \mathcal{QA}_2$ (i.e., checking whether $\mathcal{QA}_1(T) \subseteq \mathcal{QA}_2(T)$ for all T) for single-run QAs is solvable in polynomial time, since it is equivalent to checking emptiness of $\mathcal{QA}_1 \times \overline{\mathcal{QA}_2}$.

4 Logics on Trees: $\mathrm{TL}^{\mathrm{tree}}$ and XPath

$\mathbf{TL}^{\mathbf{tree}}$. An unranked tree $T = (D, \lambda)$ can be viewed as a structure $\langle D, \prec^*_{\mathrm{ch}}, \prec^*_{\mathrm{ns}}, (P_a)_{a\in\Sigma}\rangle$, where P_a's are labeling predicates: $P_a = \{s \in D \mid \lambda(s) = a\}$. Thus, when we talk about *first-order logic (*FO*)*, or *monadic second-order logic (*MSO*)*, we interpret them on these representations of unranked trees. Recall that MSO extends FO with quantification over sets.

We shall use a *tree temporal logic* [26,34], denoted here by $\mathrm{TL}^{\mathrm{tree}}$ [22]. It can be viewed as a natural extension of LTL with the past operators to unranked trees [20,38], with *next*, *previous*, *until*, and *since* operators for both child and next-sibling relations. The syntax of $\mathrm{TL}^{\mathrm{tree}}$ is defined by:

$$\varphi, \varphi' \ := \ \top \ \mid \ \bot \ \mid \ a \ \mid \ \varphi \vee \varphi' \ \mid \ \neg\varphi \ \mid \ \mathbf{X}_*\varphi \ \mid \ \mathbf{X}^-_*\varphi \ \mid \ \varphi \mathbf{U}_* \varphi' \ \mid \ \varphi \mathbf{S}_* \varphi',$$

where $\top$ and $\bot$ are true and false, a ranges over Σ, and $*$ is either 'ch' (child) or 'ns' (next sibling). The semantics is defined with respect to a tree $T = (D, \lambda)$ and a node $s \in D$:

- $(T, s) \models \top$; $\quad (T, s) \not\models \bot$;
- $(T, s) \models a$ iff $\lambda(s) = a$;
- $(T, s) \models \varphi \vee \varphi'$ iff $(T, s) \models \varphi$ or $(T, s) \models \varphi'$;
- $(T, s) \models \neg\varphi$ iff $(T, s) \not\models \varphi$;
- $(T, s) \models \mathbf{X}_{\text{ch}}\varphi$ if there exists a node $s' \in D$ such that $s \prec_{\text{ch}} s'$ and $(T, s') \models \varphi$;
- $(T, s) \models \mathbf{X}^{-}_{\text{ch}}\varphi$ if there exists a node $s' \in D$ such that $s' \prec_{\text{ch}} s$ and $(T, s') \models \varphi$;
- $(T, s) \models \varphi \mathbf{U}_{\text{ch}} \varphi'$ if there is a node s' such that $s \prec^{*}_{\text{ch}} s'$, $(T, s') \models \varphi'$, and for all $s'' \neq s'$ satisfying $s \prec^{*}_{\text{ch}} s'' \prec^{*}_{\text{ch}} s'$ we have $(T, s'') \models \varphi$.
- $(T, s) \models \varphi \mathbf{S}_{\text{ch}} \varphi'$ if there is a node s' such that $s' \prec^{*}_{\text{ch}} s$, $(T, s') \models \varphi'$, and for all $s'' \neq s'$ satisfying $s' \prec^{*}_{\text{ch}} s'' \prec^{*}_{\text{ch}} s$ we have $(T, s'') \models \varphi$.

The semantics of $\mathbf{X}_{\text{ns}}, \mathbf{X}^{-}_{\text{ns}}, \mathbf{U}_{\text{ns}}$, and $\mathbf{S}_{\text{ns}}$ is analogous by replacing the child relation with the next-sibling relation.

A TL^{tree} formula φ defines a unary query $T \mapsto \{s \mid (T, s) \models \varphi\}$. It is known that TL^{tree} is expressively complete for FO: the class of such unary queries is precisely the class of queries defined by FO formulae with one free variable [26,34].

XPath. We present a first-order complete extension of XPath, called *conditional XPath*, or CXPath [26]. We introduce very minor modifications to the syntax (e.g., we use an existential quantifier $\mathbf{E}$ instead of the usual XPath node test brackets [ ]) to make the syntax resemble that of temporal logics. CXPath has *node formulae* α and *path formulae* β given by:

$$\alpha, \alpha' := a \mid \neg\alpha \mid \alpha \vee \alpha' \mid \mathbf{E}\beta$$
$$\beta, \beta' := ?\alpha \mid \texttt{step} \mid \texttt{step}^{*} \mid (\texttt{step}/?\alpha)^{*} \mid \beta/\beta' \mid \beta \vee \beta'$$

where a ranges over Σ and $\texttt{step}$ is one of the following: $\prec_{\text{ch}}$, $\prec^{-}_{\text{ch}}$, $\prec_{\text{ns}}$, or $\prec^{-}_{\text{ns}}$. The language without the $(\texttt{step}/?\alpha)^{*}$ is known as "core XPath".

Intuitively $\mathbf{E}\beta$ states the existence of a path starting in a given node and satisfying β, the path formula $?\alpha$ tests if the node formula α is true in the initial node of a path, and / is the composition of paths.

Given a tree $T = (D, \lambda)$, the semantics of a node formula is a set of nodes $[\![\alpha]\!]_T \subseteq D$, and the semantics of a path formula is a binary relation $[\![\beta]\!]_T \subseteq D \times D$ given by the following rules. We use R^{*} to denote the reflexive-transitive closure of relation R, and $\pi_1(R)$ to denote its first projection.

$$[\![a]\!]_T = \{s \in D \mid \lambda(s) = a\} \qquad [\![?\alpha]\!]_T = \{(s, s) \mid s \in [\![\alpha]\!]_T\}$$
$$[\![\neg\alpha]\!]_T = D - [\![\alpha]\!]_T \qquad [\![\texttt{step}]\!]_T = \{(s, s') \mid s, s' \in D \text{ and } (s, s') \in \texttt{step}\}$$
$$[\![\alpha \vee \alpha']\!]_T = [\![\alpha]\!]_T \cup [\![\alpha']\!]_T \qquad [\![\beta \vee \beta']\!]_T = [\![\beta]\!]_T \cup [\![\beta']\!]_T$$
$$[\![\mathbf{E}\beta]\!]_T = \pi_1([\![\beta]\!]_T) \qquad [\![\texttt{step}^{*}]\!]_T = [\![\texttt{step}]\!]^{*}_T$$
$$[\![\beta/\beta']\!]_T = [\![\beta]\!]_T \circ [\![\beta']\!]_T$$
$$[\![(\texttt{step}/?\alpha)^{*}]\!]_T = [\![(\texttt{step}/?\alpha)]\!]^{*}_T$$

CXPath defines two kinds of unary queries: those given by node formulae, and those given by path formulae β, selecting $[\![\beta]\!]_T^{root} = \{s \in D \mid (\varepsilon, s) \in [\![\beta]\!]_T\}$. Both classes capture precisely unary FO queries on trees [26].

XPath and TL$^{\text{tree}}$. XPath expressions can be translated into TL$^{\text{tree}}$. For example, consider an expression in the "traditional" XPath syntax: $e = /a//b[//c]$. It says: start at the root, find children labeled a, their descendants labeled b, and select those which have a c-descendant. It can be viewed as both a path formula and a node formula of XPath. An equivalent path formula is

$$\beta \;= \prec_{\text{ch}} /?a/ \prec_{\text{ch}}^{*} /?(b \wedge \mathbf{E}(\prec_{\text{ch}}^{*} /?c)).$$

The set $[\![\beta]\!]_T^{root} = \{s \mid (\varepsilon, s) \in [\![\beta]\!]_T\}$ is precisely the set of nodes selected by e in T. Alternatively we can view it as a node formula

$$\alpha = b \wedge \mathbf{E}(\prec_{\text{ch}}^{*} /?c) \wedge \mathbf{E}\big((\prec_{\text{ch}}^{-})^{*}/?(a \wedge \mathbf{E}(\prec_{\text{ch}}^{-} /root))\big).$$

Here *root* is an abbreviation for a formula that tests for the root node. Then $[\![\alpha]\!]_T$ generates the set of nodes selected by e. It is known [27] that for every path formula β, one can construct in linear time a node formula α so that $[\![\beta]\!]_T^{root} = [\![\alpha]\!]_T$. Thus, from now on we deal with node XPath formulae.

The above formulae can be translated into an equivalent TL$^{\text{tree}}$ expression

$$b \wedge \mathbf{F}_{\text{ch}} c \wedge \mathbf{F}_{\text{ch}}^{-}\big(a \wedge \mathbf{X}_{\text{ch}}^{-} root\big)$$

Here $\mathbf{F}_{\text{ch}}\varphi$ is $\top \mathbf{U}_{\text{ch}}\varphi$, and $\mathbf{F}_{\text{ch}}^{-}\varphi$ is $\top \mathbf{S}_{\text{ch}}\varphi$; we also use *root* as a shorthand for $\neg \mathbf{X}_{\text{ch}}^{-} \top$. This formula selects b-labeled nodes with c-labeled descendants, and an a-ancestor which is a child of the root – this is of course equivalent to the original expression.

Since both TL$^{\text{tree}}$ and CXPath are first-order expressively-complete [26], each core or conditional XPath expression is equivalent to a formula of TL$^{\text{tree}}$; however, no direct translation has previously been produced. We now give such a direct translation that, together with the translation from TL$^{\text{tree}}$ to QAs, will guarantee single-exponential bounds on QAs equivalent to XPath formulae.

Lemma 1. *There is a translation of node formulae α of core or conditional XPath into formulae α' of* TL$^{\text{tree}}$ *such that the number of subformulae of α' is at most linear in the size of α. Moreover, if α does not use any disjunctions of path formulae, then the size of α' is at most linear in the size of α.*

In particular, even if α' is exponential in the size of α, the size of its Fischer-Ladner closure is at most linear in the size of the original formula α.

We now sketch the proof. Given two TL$^{\text{tree}}$ formulae φ and φ' and a CXPath path formula β, we write $\varphi' \equiv \mathbf{X}_\beta \varphi$ if for each tree T and each node s, one has that $(T, s) \models \varphi'$ iff there is a node s', with $(s, s') \in [\![\beta]\!]_T$, such that $(T, s') \models \varphi$. Now each CXPath node formula α is translated into a TL$^{\text{tree}}$ formula φ_α such that $(T, s) \models \varphi_\alpha$ iff $s \in [\![\alpha]\!]_T$. Each path formula β is translated into a

mapping x_β from $\mathrm{TL}^{\mathrm{tree}}$ formulae to $\mathrm{TL}^{\mathrm{tree}}$ formulae such that $x_\beta(\varphi) \equiv \mathbf{X}_\beta\varphi$. The rules are:

α	φ_α
a	a
$\neg\alpha'$	$\neg\varphi_{\alpha'}$
$\alpha' \vee \alpha''$	$\varphi_{\alpha'} \vee \varphi_{\alpha''}$
$\mathbf{E}\beta$	$x_\beta(\top)$

β	$x_\beta(\varphi)$
$?\alpha$	$\varphi_\alpha \wedge \varphi$
$\prec_{\mathrm{ch}}$	$\mathbf{X}_{\mathrm{ch}}\varphi$
$\prec_{\mathrm{ch}}^*$	$\top\mathbf{U}_{\mathrm{ch}}\varphi$
$(\prec_{\mathrm{ch}}/?\alpha)^*$	$(\mathbf{X}_{\mathrm{ch}}\varphi_\alpha)\mathbf{U}_{\mathrm{ch}}\varphi$
β'/β''	$x_{\beta'} \circ x_{\beta''}(\varphi)$
$\beta \vee \beta'$	$x_{\beta'}(\varphi) \vee x_{\beta''}(\varphi)$

□

5 Tree Logic into Query Automata: A Translation

Our goal is to translate $\mathrm{TL}^{\mathrm{tree}}$ into single-run QAs. We do a direct translation into unranked QAs, as opposed to coding of unranked trees into binary (which is a common technique). Such coding is problematic for two reasons. First, simple navigation over unranked trees may look unnatural when coded into binary, resulting in more complex formulae (child, for example, becomes 'left successor followed by zero or more right successors'). Second, coding into binary trees makes reasoning about views much harder. The property of being upward-closed, which is essential for decidability of certain answers, is not even preserved by the translation. Thus, we do a direct translation into unranked QAs, and then apply it to XML specifications.

Since values of transitions $\delta(q, a)$ in unranked QAs are not sets of states but rather NFAs representing regular languages over states, we measure the size of $\mathcal{QA} = (Q, F, Q_s, \delta)$ not as the number $|Q|$ of states, but rather as

$$\|\mathcal{QA}\| = |Q| + \sum_{q\in Q, a\in\Sigma} \|\delta(q, a)\|,$$

where $\|\delta(q, a)\|$ is the number of states of the NFA. We then show:

Theorem 1. *Every* $\mathrm{TL}^{\mathrm{tree}}$ *formula* φ *of size* n *can be translated, in exponential time, into an equivalent single-run query automaton* $\mathcal{QA}_\varphi$ *of size* $2^{O(n)}$*, i.e. a query automaton such that* $\mathcal{QA}_\varphi(T) = \{s \mid (T, s) \models \varphi\}$ *for every tree* T*.*

We now sketch the construction. First, as is common with translations into nondeterministic automata [39], we need to work with a version of $\mathrm{TL}^{\mathrm{tree}}$ in which all negations are pushed to propositions. To deal with until and since operators, we shall introduce four operators $\mathbf{R}_*$ and $\mathbf{I}_*$ for $*$ being 'ch' or 'ns' so that $\neg(\alpha\mathbf{U}_*\beta) \leftrightarrow \neg\alpha\mathbf{R}_*\neg\beta$ and $\neg(\alpha\mathbf{S}_*\beta) \leftrightarrow \neg\alpha\mathbf{I}_*\neg\beta$; this part is completely standard. However, trees do not have a linear structure and we cannot just push negation inside the $\mathbf{X}$ operators: for example, $\neg\mathbf{X}_{\mathrm{ch}}\varphi$ is not $\mathbf{X}_{\mathrm{ch}}\neg\varphi$. Since our semantics of the next operators is existential (there is a successor node in which the formula is true), we need to add their universal analogs. For example, $\mathbf{X}^\forall_{\mathrm{ch}}\varphi$

is true in s if for every successor s' of s in the domain of the tree, φ is true in s'. Then of course we have $\neg \mathbf{X}_{\mathrm{ch}} \varphi \leftrightarrow \mathbf{X}_{\mathrm{ch}}^{\forall} \neg \varphi$. We add four such operators $(\mathbf{X}_{\mathrm{ch}}^{\forall}, \mathbf{X}_{\mathrm{ns}}^{\forall}, \mathbf{X}_{\mathrm{ch}}^{-\forall}, \mathbf{X}_{\mathrm{ns}}^{-\forall})$. Other axes have a linear structure, so one could alternatively add tests for the root, first, and last child of a node to deal with them. For example, $\neg \mathbf{X}_{\mathrm{ch}}^{-} \varphi \leftrightarrow \mathbf{X}_{\mathrm{ch}}^{-} \neg \varphi \vee \alpha_{root}$, where α_{root} is a test for the root. But for symmetry we prefer to deal with the four universal versions of the next/previous operators, since it is unavoidable for $\mathbf{X}_{\mathrm{ch}}$.

With these additions, we can push negations to propositions, so we assume negations only occur in subformulae $\neg a$ for $a \in \Sigma$. The states of $\mathcal{QA}_{\varphi}$ will be maximally consistent subsets of the Fischer-Ladner closure of φ (in particular, for each state q and a subformula ψ, exactly one of ψ and $\neg\psi$ is in q).

The transitions have to ensure that all "horizontal" temporal connectives behave properly, and that "vertical" transitions are consistent. The alphabet of each automaton $\delta(q, a)$ is the set of states of $\mathcal{QA}_{\varphi}$; that is, letters of $\delta(q, a)$ are sets of formulae. Each $\delta(q, a)$ is a product of three automata. The first guarantees that eventualities $\alpha \mathbf{U}_{\mathrm{ns}} \beta$ and $\alpha \mathbf{S}_{\mathrm{ns}} \beta$ are fulfilled in the oldest and youngest siblings. For that, we impose conditions on the initial states $\delta(q, a)$'s that they need to read a letter (which is a state of $\mathcal{QA}_{\varphi}$) that may not contain $\alpha \mathbf{S}_{\mathrm{ns}} \beta$ without containing β, and on their final state guaranteeing that in the last letter we do not have a subformula $\alpha \mathbf{U}_{\mathrm{ns}} \beta$ without having β.

The second automaton enforces horizontal transitions, and it behaves very similarly to the standard LTL-to-Büchi construction; it only deals with next-sibling connectives. For example, if $\mathbf{X}_{\mathrm{ns}} \alpha$ is the current state of $\mathcal{QA}$ for a node $s \cdot i$, then the state for $s \cdot (i+1)$ contains α, and that if $\alpha \mathbf{U}_{\mathrm{ns}} \beta$ is in the state for $s \cdot i$ but β is not, then $\alpha \mathbf{U}_{\mathrm{ns}} \beta$ is propagated into the state for $s \cdot (i+1)$.

The third automaton enforces vertical transitions. We give a few sample rules. If q contains the negation of $\alpha \mathbf{S}_{\mathrm{ch}} \beta$, then the automaton rejects after seeing a state which contains $\alpha \mathbf{S}_{\mathrm{ch}} \beta$ but does not contain β (since in this case $\alpha \mathbf{S}_{\mathrm{ch}} \beta$ must propagate to the parent). If q contains $\alpha \mathbf{U}_{\mathrm{ch}} \beta$ and does not contain β, then the automaton only accepts if one of its input letters contains $\alpha \mathbf{U}_{\mathrm{ch}} \beta$. And if q contains $\mathbf{X}_{\mathrm{ch}} \alpha$, then it only accepts if one of its input letters contains α. In addition, we have to enforce eventualities $\alpha \mathbf{U}_{\mathrm{ch}} \beta$ by disallowing these automata to accept ε if q contains $\alpha \mathbf{U}_{\mathrm{ch}} \beta$ and does not contain β.

The final states of $\mathcal{QA}_{\varphi}$ at the root must enforce correctness of $\alpha \mathbf{S}_{\mathrm{ch}} \beta$ formulae: with each such formula, states from F must contain β as well. This completes the construction. When all automata $\delta(q, a)$ are properly coded, the $2^{O(n)}$ bound follows. We then show a standard lemma that in an accepting run, a node is assigned a state that contains a subformula α iff α is true in that node. This guarantees that for every tree, there is an accepting run. Since each state has either α or $\neg\alpha$ in it, it follows that the resulting QA is single-run.

6 An Application: Reasoning about Document Navigation

As mentioned in Section 2, typical XML static analysis tasks include consistency of schema and navigational properties (e.g., is a given XPath expression

consistent with a given DTD?), or query optimization (e.g., is a given XPath expression e contained in a another expression e' for all trees that conform to a DTD d?). We now show two applications of our results for such analyses of XML specifications.

Satisfiability algorithms for sets of XPath expressions. The exponential-time complexity for satisfiability of XPath expressions in the presence of a schema is already known [25,6]. We now show how we can verify satisfiability of multiple sets of XPath expressions, in a uniform way, using translation into query automata.

Given an arbitrary set $E = \{e_1, \ldots, e_n\}$ of XPath (core or conditional) expressions and a subset $E' \subseteq E$, let $\mathcal{Q}(E')$ be a unary query defining the intersection of queries given by all the $e \in E'$. That is, $\mathcal{Q}(E')$ selects nodes that satisfy every expression $e \in E'$. We can capture *all* (exponentially many) such queries $\mathcal{Q}(E')$s by a single automaton, that is instantiated into different QAs by different selecting states.

Corollary 1. *One can construct, in time* $2^{O(\|E\|)}$ *(that is,* $2^{O(\|e_1\|+\ldots+\|e_n\|)}$*), an unranked tree automaton* $\mathcal{A}(E) = (Q, F, \delta)$ *and a relation* $\sigma \subseteq E \times Q$ *such that, for every* $E' \subseteq E$*,*

$$\mathcal{QA}_{E'} \;=\; (Q,\ F,\ \bigcap\{\sigma(e) \mid e \in E'\}, \delta)$$

is a single-run QA defining the unary query $\mathcal{Q}(E')$*.*

The construction simply takes the product of all the $\mathcal{QA}_{e'_i}$s, produced by Theorem 1, where e'_i is a $\mathrm{TL}^{\mathrm{tree}}$ translation of e_i, produced by Lemma 1. The relation σ relates tuples of states that include selecting states of $\mathcal{QA}_{e'_i}$ with $e_i \in E$. Then checking nonemptiness of $\mathcal{QA}_{E'}$, we see if all $e \in E'$ are simultaneously satisfiable.

The containment problem for XPath expressions is a special case of the problem we consider. To check whether $d \models e_1 \subseteq e_2$, we construct $\mathcal{QA}_{\{e_1, \neg e_2\}}$ as in Corollary 1, and take the product of it with the automaton for d. This results in a QA of size $\|d\| \cdot 2^{O(\|e_1\|+\|e_2\|)}$ that finds counterexamples to containment under d. This is precisely the construction that was promised in the introduction.

Verifying complex containment statements under DTDs. We can now extend the previous example and check not a single containment, as is usually done [35], but arbitrary Boolean combinations of XPath containment statements, without additional complexity. Assume that we are given a DTD d (or any other schema specification presented by an automaton), a set $\{e_1, \ldots, e_n\}$ of XPath expressions, and a Boolean combination $\mathcal{C}$ of inclusions $e_i \subseteq e_j$. We now want to check whether $d \models \mathcal{C}$, that is, whether $\mathcal{C}$ is true in every tree T that conforms to d. We shall refer to size of $\mathcal{C}$ as $\|\mathcal{C}\|$; the definition is extended in the natural way from the definition of $\|e\|$.

Theorem 2. *In the above setting, one can construct an unranked tree automaton of size* $\|d\| \cdot 2^{O(\|\mathcal{C}\|)}$ *whose language is empty iff* $d \models \mathcal{C}$*.*

This is achieved by replacing $e_i \subseteq e_j$ in $\mathcal{C}$ with the formula $\neg\mathbf{F}_{\mathrm{ch}}(e_i' \wedge \neg e_j')$ and $e_i \not\subseteq e_j$ in $\mathcal{C}$ with the formula $\mathbf{F}_{\mathrm{ch}}(e_i' \wedge \neg e_j')$, where e_i', e_j' are $\mathrm{TL}^{\mathrm{tree}}$ translations of e_i and e_j produced by Lemma 1. Thus we can view $\mathcal{C}$ as a $\mathrm{TL}^{\mathrm{tree}}$ formula $\alpha_{\mathcal{C}}$. Now construct a QA for $\neg\alpha_{\mathcal{C}}$, by Theorem 1, and turn it into an automaton that checks whether the root gets selected. Now we take the product of this automaton with the automaton for d. The result accepts counterexamples to $\mathcal{C}$ under d, and the result follows. The construction of the automaton is polynomial-time in $\|d\|$ and single-exponential time in $\|\mathcal{C}\|$.

7 An Application: Reasoning about Views

Recall the problem outlined in the introduction. We have a view definition given by a query automaton $\mathcal{QA}_{\mathcal{V}}$. For each source tree T, it selects a set of nodes $V = \mathcal{QA}_{\mathcal{V}}(T)$ which can also be viewed as a tree (we can assume, for example, that $\mathcal{QA}_{\mathcal{V}}$ always selects the root). Source trees are required to satisfy a schema constraint (e.g., a DTD). Since all schema formalisms for XML are various restrictions or reformulations of tree automata, we assume that the schema is given by an automaton $\mathcal{A}$.

If we only have access to V, we would like to be sure that secret information about an unknown source T is not revealed. This information, which we assume to be coded by a Boolean query $\mathfrak{Q}$, would be revealed by V if the answer to $\mathfrak{Q}$ were true in all source trees T that conform to the schema and generate V – that is, if $\underline{certain}^{\mathcal{A}}_{\mathcal{QA}_{\mathcal{V}}}(\mathfrak{Q}; V)$ were true. Thus, we would like to construct a new automaton $\mathcal{A}^*$ that accepts V iff $\underline{certain}^{\mathcal{A}}_{\mathcal{QA}_{\mathcal{V}}}(\mathfrak{Q}; V)$ is false, giving us some security assurances about the view.

In general, such an automaton construction is impossible: if $\mathcal{QA}_{\mathcal{V}}$ generates the yield of a tree, views essentially code context-free languages. Combining multiple CFLs with the help of DTDs, we get an undecidability result:

Proposition 1. *The problem of checking, for source and view schemas $\mathcal{A}_s$ and $\mathcal{A}_v$, a view definition $\mathcal{QA}_{\mathcal{V}}$, and a Boolean first-order query $\mathfrak{Q}$, whether there exists a view V that conforms to $\mathcal{A}_v$ and satisfies $\underline{certain}^{\mathcal{A}_s}_{\mathcal{QA}_{\mathcal{V}}}(\mathfrak{Q}; V) = \mathrm{true}$, is undecidable.*

Schemas and queries required for this result are very simple, so to ensure the existence of the automaton $\mathcal{A}^*$, we need to put restrictions on the class of views. We assume that they are *upward-closed* as in [7]: if a node is selected, then so is the entire path to it from the root.

Note that the upward-closure $\mathcal{QA}_{\uparrow}$ of a query automaton $\mathcal{QA}$ can be obtained in linear time by adding a bit to the state indicating whether a selecting state has been seen and propagating it up. Thus, we shall assume without loss of generality that QAs defining views are *upward-closed*: if $s \in \mathcal{QA}(T)$ and s' is an ancestor of s, then $s' \in \mathcal{QA}(T)$.

The key observation that we need is that for an upward-closed QA, satisfying the single-run condition, its image is regular. Furthermore, it can be accepted by a small tree automaton:

Lemma 2. *Let $\mathcal{QA}$ be an upward-closed query automaton that satisfies condition 1) of the definition of single-run QAs. Then one can construct, in cubic time, an unranked tree automaton $\mathcal{A}^*$ that accepts trees V for which there exists a tree T satisfying $V = \mathcal{QA}(T)$. Moreover, the number of states of $\mathcal{A}^*$ is at most the number of states of $\mathcal{QA}$.*

Proof sketch. The automaton $\mathcal{A}^*$ has to guess a tree T and its run so that the selecting states would be assigned precisely to the elements of V. So one first needs to analyze non-selecting runs: that is, runs that can be extended to an accepting run but never hit a selecting state. Trees admitting such runs may be inserted under leaves of V, and in between two consecutive siblings of a node in V. We then need to modify the horizontal transition to allow for guesses of words consisting of final states of non-selecting runs in between two states. □

To apply Lemma 2 to the problem of finding certain answers $\underline{certain}^{\mathcal{A}}_{\mathcal{QA}_\mathcal{V}}(\mathfrak{Q}; V)$, we now take the product of $\mathcal{QA}_\mathcal{V}$ with $\mathcal{A}$ and the automaton for $\neg\mathfrak{Q}$ (the selecting states in the product will be determined by $\mathcal{QA}_\mathcal{V}$), and obtain:

Theorem 3. *Let $\mathcal{QA}_\mathcal{V}$ be upward-closed and single-run, $\mathcal{A}$ an unranked tree automaton defining a schema, and $\mathcal{A}_{\neg\mathfrak{Q}}$ an automaton accepting trees for which $\mathfrak{Q}$ is false. Then one can construct, in polynomial time, an unranked tree automaton $\mathcal{A}^*$ such that*

1. $\|\mathcal{A}^*\| = O(\|\mathcal{QA}_\mathcal{V}\| \cdot \|\mathcal{A}\| \cdot \|\mathcal{A}_{\neg\mathfrak{Q}}\|)$, *and*
2. $\mathcal{A}^*$ accepts $V \quad \Leftrightarrow \quad \underline{certain}^{\mathcal{A}}_{\mathcal{QA}_\mathcal{V}}(\mathfrak{Q}; V) = \text{false}$.

Combining Theorem 3 with previous translations into single-run QAs and properties of the latter, we obtain algorithms for verifying properties of views given by XPath expressions. Revisiting our motivating example from Section 2, we make the following assumptions:

- The view definition is given by an XPath (conditional or core) expression e_V; the view V of a source tree T has all the nodes selected by e_V and their ancestors;
- The schema definition is given by a DTD d;
- The query $\mathfrak{Q}$ is an arbitrary Boolean combination of containment statements $e \subseteq e'$, where e, e' come from a set E of XPath expressions.

Then, for a given V, we want to check if $\underline{certain}^{d}_{e_V}(\mathfrak{Q}; V)$ is false: that is, the secret encoded by $\mathfrak{Q}$ cannot be revealed by V, since not all source trees T that conform to d and generate V satisfy $\mathfrak{Q}$. We then have the following:

Corollary 2. *In the above setting, one can construct in time polynomial in $\|d\|$ and exponential in $\|E\| + \|e_V\|$ an unranked tree automaton $\mathcal{A}^*$ of size $\|d\| \cdot 2^{O(\|e_V\|+\|E\|)}$ that accepts a view V iff $\underline{certain}^{d}_{e_V}(\mathfrak{Q}; V)$ is false.*

Note that again the exponent contains the size of typically small XPath expressions, and not the potentially large schema definition d.

8 Conclusion

There are several extensions we would like to consider. One concerns relative specifications often used in the XML context – these apply to subtrees. Results of [21,2] on model-checking of *now* and *within* operators on words and nested words indicate that an exponential blowup is unavoidable, but there could well be relevant practical cases that do not exhibit it. We would like to see how LTL-to-Büchi optimization techniques (e.g., in [12,17]) could be adapted in our setting, to produce automata of smaller size. We also would like to see if automata can be used for reasoning about views without imposing upward-closeness of [7], which does not account for some of the cases of secure XML views [13]. One could look beyond first-order at logics having the power of MSO or ambient logics with known translations into automata, and investigate their translations into QAs [9,18,15]. Another possible direction has to do with a SAX representation of XML which corresponds to its linear structure (in the paper we dealt with the tree structure, i.e., the DOM representation). The connection between the linear structure of XML and nested words already found some applications [19,23].

Acknowledgment. We thank Pablo Barceló and Floris Geerts for their comments. This work was done while the second author was at the University of Edinburgh. The authors were supported by EPSRC grant E005039, the first author also by the European Commission Marie Curie Excellence grant MEXC-CT-2005-024502.

References

1. Abiteboul, S., Cautis, B., Milo, T.: Reasoning about XML update constraints. In: PODS 2007, pp. 195–204 (2007)
2. Alur, R., Arenas, M., Barceló, P., Etessami, K., Immerman, N., Libkin, L.: First-order and temporal logics for nested words. In: LICS 2007, pp. 151–160 (2007)
3. Alur, R., Etessami, K., Madhusudan, P.: A temporal logic of nested calls and returns. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 467–481. Springer, Heidelberg (2004)
4. Arenas, M., Fan, W., Libkin, L.: Consistency of XML specifications. In: Inconsistency Tolerance, pp. 15–41. Springer, Heidelberg (2005)
5. Barceló, P., Libkin, L.: Temporal logics over unranked trees. In: LICS 2005, pp. 31–40 (2005)
6. Benedikt, M., Fan, W., Geerts, F.: XPath satisfiability in the presence of DTDs. In: PODS 2005, pp. 25–36 (2005)
7. Benedikt, M., Fundulaki, I.: XML subtree queries: specification and composition. In: Bierman, G., Koch, C. (eds.) DBPL 2005. LNCS, vol. 3774, pp. 138–153. Springer, Heidelberg (2005)
8. Bojanczyk, M., David, C., Muscholl, A., Schwentick, T., Segoufin, L.: Two-variable logic on data trees and XML reasoning. In: PODS 2006, pp. 10–19 (2006)
9. Boneva, I., Talbot, J.-M., Tison, S.: Expressiveness of a spatial logic for trees. In: LICS 2005, pp. 280–289 (2005)

10. Calvanese, D., De Giacomo, G., Lenzerini, M., Vardi, M.Y.: Regular XPath: constraints, query containment and view-based answering for XML documents. In: Logic in Databases (2008)
11. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
12. Daniele, M., Giunchiglia, F., Vardi, M.Y.: Improved automata generation for linear temporal logic. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 249–260. Springer, Heidelberg (1999)
13. Fan, W., Geerts, F., Jia, X., Kementsietsidis, A.: Rewriting regular XPath queries on XML views. In: ICDE 2007, pp. 666–675 (2007)
14. Fan, W., Chan, C.Y., Garofalakis, M.: Secure XML querying with security views. In: SIGMOD 2004, pp. 587–598 (2004)
15. Frick, M., Grohe, M., Koch, C.: Query evaluation on compressed trees. In: LICS 2003, pp. 188–197 (2003)
16. Genevés, P., Layaida, N.: A system for the static analysis of XPath. ACM TOIS 24, 475–502 (2006)
17. Gerth, R., Peled, D., Vardi, M., Wolper, P.: Simple on-the-fly automatic verification of linear temporal logic. In: PSTV 1995, pp. 3–18 (1995)
18. Gottlob, G., Koch, C.: Monadic datalog and the expressive power of languages for web information extraction. J. ACM 51, 74–113 (2004)
19. Kumar, V., Madhusudan, P., Viswanathan, M.: Visibly pushdown automata for streaming XML. In: WWW 2007, pp. 1053–1062 (2007)
20. Kupferman, O., Pnueli, A.: Once and for all. In: LICS 1995, pp. 25–35 (1995)
21. Laroussinie, F., Markey, N., Schnoebelen, P.: Temporal logic with forgettable past. In: LICS 2002, pp. 383–392 (2002)
22. Libkin, L.: Logics for unranked trees: an overview. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 35–50. Springer, Heidelberg (2005)
23. Madhusudan, P., Viswanathan, M.: Query automata for nested words (manuscript, 2008)
24. Maneth, S., Perst, T., Seidl, H.: Exact XML type checking in polynomial time. In: Schwentick, T., Suciu, D. (eds.) ICDT 2007. LNCS, vol. ICDT 2007, pp. 254–268. Springer, Heidelberg (2006)
25. Marx, M.: XPath with conditional axis relations. In: Bertino, E., Christodoulakis, S., Plexousakis, D., Christophides, V., Koubarakis, M., Böhm, K., Ferrari, E. (eds.) EDBT 2004. LNCS, vol. 2992, pp. 477–494. Springer, Heidelberg (2004)
26. Marx, M.: Conditional XPath. ACM TODS 30, 929–959 (2005)
27. Marx, M., de Rijke, M.: Semantic characterizations of navigational XPath. SIGMOD Record 34, 41–46 (2005)
28. Neven, F.: Design and Analysis of Query Languages for Structured Documents. PhD Thesis, U. Limburg (1999)
29. Neven, F.: Automata, logic, and XML. In: Bradfield, J.C. (ed.) CSL 2002 and EACSL 2002. LNCS, vol. 2471, pp. 2–26. Springer, Heidelberg (2002)
30. Neven, F., Schwentick, T.: Query automata over finite trees. TCS 275, 633–674 (2002)
31. Neven, F., Schwentick, T.: On the complexity of XPath containment in the presence of disjunction, DTDs, and variables. LMCS 2(3) (2006)
32. Neven, F., Van den Bussche, J.: Expressiveness of structured document query languages based on attribute grammars. J. ACM 49(1), 56–100 (2002)

33. Niehren, J., Planque, L., Talbot, J.-M., Tison, S.: N-ary queries by tree automata. In: Bierman, G., Koch, C. (eds.) DBPL 2005. LNCS, vol. 3774, pp. 217–231. Springer, Heidelberg (2005)
34. Schlingloff, B.-H.: Expressive completeness of temporal logic of trees. Journal of Applied Non-Classical Logics 2, 157–180 (1992)
35. Schwentick, T.: XPath query containment. SIGMOD Record 33, 101–109 (2004)
36. Schwentick, T.: Automata for XML – a survey. JCSS 73, 289–315 (2007)
37. Vardi, M.Y.: An automata-theoretic approach to linear temporal logic. In: Banff Higher Order Workshop (1996)
38. Vardi, M.Y.: Reasoning about the past with two-way automata. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443, pp. 628–641. Springer, Heidelberg (1998)
39. Vardi, M.Y., Wolper, P.: Reasoning about infinite computations. Inf.& Comput. 115, 1–37 (1994)
40. Vardi, M.Y., Wolper, P.: Automata-theoretic techniques for modal logics of programs. JCSS 33, 183–221 (1986)

Distributed Consistency-Based Diagnosis

Vincent Armant, Philippe Dague, and Laurent Simon

LRI, Univ. Paris-Sud 11, CNRS and INRIA Saclay
Parc Club Université, 4 rue Jacques Monod 91893 Orsay Cedex, France
{vincent.armant,philippe.dague,laurent.simon}@lri.fr

Abstract. A lot of methods exist to prevent errors and incorrect behaviors in a distributed framework, where all peers work together for the same purpose, under the same protocol. For instance, one may limit them by replication of data and processes among the network. However, with the emergence of web services, the willing for privacy, and the constant growth of data size, such a solution may not be applicable. For some problems, failure of a peer has to be detected and located by the whole system. In this paper, we propose an approach to diagnose abnormal behaviors of the whole system by extending the well known consistency-based diagnosis framework to a fully distributed inference system, where each peer only knows the existence of its neighbors. Contrasting with previous works on model-based diagnosis, our approach computes all minimal diagnoses in an incremental way, without needs to get any conflict first.

1 Introduction

Model-Based Diagnosis has been introduced in the late eighties by [14,11], and has since been widely used in many successful works. With this formalism, a logical theory describes the normal (and, optionally, abnormal) behavior of a physical system, and consistency checking against observations is used to derive hypotheses over components reliability (called *diagnoses*), that explain failures. Even if stronger logic may be used, it is often the case where propositional logic is chosen to model the system. In this context, diagnosing the system with respect to observations can be expressed as a classical – and heavily studied – knowledge based compilation problem: restricted prime implicants [6].

Recent years have seen an increasing number of AI works pushing forward the power of *distributed systems*, for instance by adding semantic layers [1]. In such networks, all systems (or "peers") are running the same algorithm, and are working for the same purpose. The framework may however describe two kinds of settings. One which allows any peer to communicate with any other peer (generally by means of distributed hash tables, [17]) and the other where peers only know their neighbors, which is closer to social networks, circuits, and web services composition. In the latter formalism, reasoning is based on the declaration of logical equivalence of variables between peers (the *shared variables*), which locally defines *subsystems* acquaintances.

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 113–127, 2008.

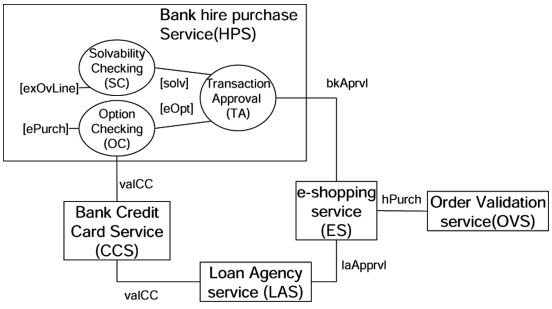


Fig. 1. 3 steps web-payment certification

In this paper, we investigate the problem of diagnosing distributed systems defined by peers acquaintances. Each peer only knows its neighborhood, and has a logical model of its normal and abnormal behavior with respect to its own local variables and its variables shared with its acquaintances, only some of them are observable. The challenging problem is to build a set of global diagnoses for the whole system. Our solution directly computes diagnoses (including all minimal ones for set inclusion) without conflicts analysis, a very hard task which is generally the first step – and the first bottleneck – of all previous model-based diagnoses engines, even when efficient algorithms are used [16].

In our approach, we focus on "static" settings of distributed systems (i.e. we do not deal with connection of new peers or disconnection of existing peers), in order to easily ensure that diagnoses and observations are consistent. If the static behavior is not possible in a fully peer-to-peer setting, it is more realistic in a distributed setting, for instance web services composition, embedded circuits, and social networks. In many cases, additional layers, like memory of past events and counters, can even simulate the "static" hypothesis.

In the next section, we introduce our notations, recall the principles of model based diagnosis and extend it to formulas in Disjunctive Normal Form. In section 3, we introduce our foundations of distributed reasoning for diagnosis. In section 4, we present the distributed algorithm and then we report related work and conclude.

Example 1 (Three steps web-payment certification). We illustrate the paper by a toy example of a web-payment certification, see figure 1. The order validation service (OVS) asks to an eshopping service (ES) for a hire purchase approval (hPurch). In order to maximize its sales opportunity, (ES) waits for the customer bank approval (bkAprvl) or a loan agency approval (laAprvl). The bank hire purchase service (HPS) and the loan agency service (LAS) both check the customer credit card validity (valCC) by a call to the credit card service (CCS). In the following, we restrict the system to (HPS) and will refer to its "global description" as the conjunction of the Transaction Approval (TA), the Solvability Checking (SC) and the Option Checking (OC).

2 From CNF Diagnosis to DNF Diagnosis

We assume familiarity with the standard literature on propositional reasoning and resolution. A literal is a variable v or its negation $\neg v$. Given a set L of literals, we denote by $\overline{L}$ the set of its opposite literals. A Conjunctive Normal Form formula (CNF) is a conjunction of clauses (disjunctions of literals). A Disjunctive Normal Form formula (DNF) is a disjunction of products (conjunctions of literals). For simplicity, we identify a formula with a set of sets of literals. We denote by $T^{\wedge}$ (resp $T^{\vee}$) the set of sets of literals corresponding to a CNF (resp DNF).

A *model* is a set of literals that, when assigned to true, allows the evaluation of a given formula to true. We say that a formula f is satisfiable (or consistent), denoted by $f \not\models \bot$, if there exists a model for f. Let f_1, and f_2 be two formulas, if all the models of f_1 are also models of f_2, which is noted $f_1 \models f_2$, then f_1 is called an *implicant* of f_2 and f_2 is called an *implicate* of f_1. The minimal (with respect to the order relation induced by inclusion of sets of literals) implicate clauses (resp. implicant products) of a formula are called the prime implicates (resp. prime implicants) of this formula. The set of prime implicates is expressed in CNF whether the set of prime implicants is in DNF. Given a formula f and a subset V of its variables, the restriction of f on V is denoted by $f|_V$ and corresponds to recursively apply on f the Shannon decomposition operator on all variables x of f that do not appear in V. This operation, known as *forgetting* in Knowledge Compilation, is well known to be a NP-Hard problem. However, when f is expressed as a DNF, the restriction operator is simply a vocabulary restriction of all products of f. The restriction of $T^{\vee}$ on a set of literals L can be defined as $T^{\vee}|_{\{L\}} = \{I|_{\{L\}} | \exists I \in T^{\vee}$ s.t. $I|_{\{L\}} = I \cap L\}$, which is no more a hard task.

2.1 Centralized Model-Based Diagnosis

Like many other works, we adapt the model-based diagnosis framework from [11,8] to the propositional case. Initially, an observed system is a triple (SD, COMPS, OBS) where SD is a first order logical formula describing the system behavior, OBS is a formula describing the observations (that boils down frequently to values assignment to observable variables) and COMPS is the set of monitored components, that appear as arguments of the predefined predicate $Ab()$ in SD ($Ab(C_i)$ denoting that component C_i is abnormal). In propositional logic, we may merge the whole into a single theory T, with the naming convention: all variables okC_i (called mode variables) encode the correct behavioral modes of the components C_i, i.e. $\neg Ab(C_i)$. We note F the set of negative mode literals {...,$\neg okC_i$,...} representing faulty components. For a (boolean) observable coded by a variable v, the elementary observation $v = a$ is coded by v if a equals 1 and $\neg v$ if a equals 0.

Example 2 (Modeling the system). A correct behavior of TA ($okTA$) will approve a hire purchase ($bkAprvl$) if the customer is solvent ($solv$) and fulfills the condition ($eOpt$) of OC. The rule for TA is rewritten as $f(TA) : okTA \Rightarrow (solv \wedge eOpt \Leftrightarrow bkAprvl)$. A normal functioning of SC ($okSC$) will consider a customer

solvent (solv) if he does not exceed his overdraft limit ($\neg exOvLine$). We obtain $f(SC) : okSC \Rightarrow (\neg exOvLine \Leftrightarrow solv)$. A correct behavior of OC (okOC) will satisfy (eOPt) if the customer asked for hire purchases by internet (ePurch) and his credit card is valid (valCC). There are only two possible failures for OC: when ePurch keeps its default value (i.e. no internet purchase) whereas the customer asked for the internet option, and when the customer card is believed invalid whereas it is valid. The Option Checking system can thus be encoded by $f(OC) : okOC \Rightarrow (ePurch \wedge valCC \Leftrightarrow eOpt) \wedge (\neg okOC \Rightarrow \neg valCC \vee \neg ePurch)$. The behavior of HPS is the conjunction $f(HPS) : f(OC) \wedge f(SC) \wedge f(TA)$.

A diagnosis is a behavioral mode assignment to each component of the system, consistent with its theory.

Definition 1 (Minimal Diagnoses). *Let T be the theory that describes an observed system, F the consistent set of all negative mode literals of the system.*

$$\triangle \subseteq F \text{ is a diagnosis for } T \text{ iff } T \cup \triangle \cup \overline{\{F \setminus \triangle\}} \not\models \bot$$

We write $Diag(T)$ the set of diagnoses of T and $min_{\subseteq}(Diag(T))$ the set of its minimal diagnoses.

Intuitively, this definition states that, given any minimal diagnosis $\triangle$, one may suppose that all components C' that do not appear in $\triangle$ are correct. We may also notice that, because we can restrict the set of possible failures ($(\neg okOC \Rightarrow \neg valCC \vee ePurch)$ in the previous example), a minimal diagnosis may not be extended by supposing some component C' incorrect (a negative mode literal candidate for extending a diagnosis can be inconsistent with T and the other mode literals).

The theorem 3 of [8] states that the minimal diagnoses are the prime implicants of the conjunction of minimal conflicts, where the minimal conflicts (called minimal conflict sets in [14] and minimal positive conflicts in [8]) are the prime implicates of the formula $SD \wedge OBS$, restricted to the mode literals in F. Intuitively, a minimal conflict refers to a set of components containing at least a faulty one. Minimal diagnoses are thus the minimal (for literals set inclusion) conjunctions of faulty components that can explain all the conflicts, according to observations.

Example 3 (Conflicts and Diagnoses on Scenario 1). Let us suppose the following scenario: the bank approved a hire purchase for an operation whereas the customer exceeds his overdraft limit. He had a valid credit card but asked to stop internet purchasing. In this case the bank service does not fulfill its expected behavior.

We thus look for the minimal subsets of {TA, OC, SC} that may be faulty. This will be expressed by minimal conjunctions of literals from {$\neg okTA$,$\neg okOC$,$\neg okSC$}, which are consistent with the formula $f(HPS)$ and the observations {exOvLine, valCC, $\neg ePurch$, bkAprvl }. The minimal conflicts are ($\neg okSC \vee \neg okTA$) and ($\neg okOC \vee \neg okTA$). The minimal diagnoses that satisfy these conflicts are: ($\neg okTA$) and ($\neg okSC \wedge \neg okOC$).

Most of previous works on diagnosis compute first the set of conflicts, restricted to mode literals. Then, only when this first stage is over, diagnoses can be computed. These methods are hopeless for building an incremental diagnostic engine

as all minimal conflicts have to be known before the first diagnosis can be returned. They are nevertheless motivated by the fact that models of real-world systems are supposed to be close to CNF. If needed, new variables are usually added to practically contain the potential blow-up when translating to CNF. Up to now very few interest has been shown in DNF representations of a system. However, such a DNF representation is very advantageous for diagnosis: if we ensure that the set F of mode variables is consistent, which means that no variable appears both positively and negatively in F, then, if $T^\vee$ is the description of an observed system, each product of the restriction $T^\vee|_{\{F\}}$ is a diagnosis (not necessarily minimal).

Lemma 1. *Let $T^\vee$ be a DNF description of an observed system, and F a consistent set of negative mode literals, then*

$$\forall I \in T^\vee, I|_{\{F\}} \in Diag(T)$$

Sketch of Proof. For each $I \in T^\vee$, I is an implicant of T which is trivially consistent with T. Let us consider $\{F \backslash I|_{\{F\}}\}$, which does not contain any literals from $I|_{\{F\}}$. Thus, $T \cup I|_{\{F\}} \cup \{F \setminus I|_{\{F\}}\}$ is consistent and by definition $I|_{\{F\}}$ is a diagnosis.

Consequently, if we compute at least one implicant of T, we obtain at least one diagnosis without waiting for conflicts. In practice, our requirement about a DNF representation of T can be weakened without loss: implicants can be incrementally computed by an efficient SAT solver or even deduced from a compact, compiled, representation of T that allows efficient production of models [6]. The result is that, on small systems, or on large – but distributed – systems, the direct translation from CNF to DNF can be done. The following theorem states that minimal diagnoses are contained in any DNF description encoding the observed system.

Theorem 1. *Let T be the description of an observed system, and F a consistent set of negative mode literals:*

$$min_\subseteq(Diag(T)) = min_\subseteq(T^\vee|_{\{F\}})$$

Sketch of Proof A) Let $\triangle$ be a minimal diagnosis, by the definition 1, $\triangle \cup \overline{\{F \setminus \triangle\}}$ is consistent with the observed system. Thus, for any DNF representation of the system, there exists an implicant I consistent with $\triangle \cup \overline{\{F \setminus \triangle\}}$. Since I is consistent with $\triangle \cup \overline{\{F \setminus \triangle\}}$ we have $I|_{\{F \setminus \triangle\}} = \emptyset$ and thus $I|_{\{F\}}$ $=I|_{\{\triangle\}}$. Because we know that $I|_{\{F\}}$ is a diagnosis and $\triangle$ *is a minimal one* we have $I|_{\{\triangle\}} = \triangle$. **B**) Let $I|_{\{F\}} \in min_\subseteq(T^\vee|_{\{F\}})$, $I|_{\{F\}}$ is a diagnosis, suppose that it is not a minimal one. Then there exists a diagnosis $\triangle$, s.t. $\triangle \subset I|_{\{F\}}$. Consequently, there exists l in $I|_{\{F\}}$ s.t. l is not in $\triangle$. In this case $\triangle \cup \overline{\{ F \setminus \triangle\}} \cup I$ is contradictory. But since $\triangle \cup \overline{\{ F \setminus \triangle\}}$ is consistent with $T^\vee$, then there exists $I' \neq I$ s.t. $\triangle \cup \overline{\{ F \setminus \triangle\}} \cup I'$ is consistent and $I'|_{\{F\}}$ $\subseteq \triangle$. We deduce that $I'|_{\{F\}} \subseteq \triangle \subset I|_{\{F\}}$. It is contradictory with the fact that $I|_{\{F\}} \in min_\subseteq(T^\vee|_{\{F\}})$.

Example 4 (Finding Diagnoses in DNF theory). Let $F_{HPS} = \{\neg okTA, \neg okOC, \neg okSC\}$ *be the set of mode literals and* $T^{\vee}_{HPS}$ *the DNF formula of the description of HPS with observations.*

$$T^{\vee}_{HPS} = \begin{array}{l}(\neg okTA \wedge \neg okSC \wedge \neg eOpt \wedge okOC) \vee \\ (\neg okTA \wedge \neg okSC \wedge \neg okOC) \vee \\ (\neg okTA \wedge \neg solv \wedge \neg okOC) \vee \\ (\neg okTA \wedge \neg okSC \wedge \neg eOpt) \vee \\ (\neg okTA \wedge \neg solv \wedge \neg eOpt) \vee \\ (\neg okSC \wedge \neg okOC \wedge eOpt \wedge solv)\end{array}$$

For simplicity, we omitted, in each product, the conjunction of observed literals $exOvLine \wedge valCC \wedge \neg ePurch \wedge bkAprvl$. *Finally, after restriction on* F_{HPS} *and subsumption elimination, we obtain the two diagnoses* $\{\neg okTA, (\neg okSC \wedge \neg okOC)\}$.

By the lemma 1 we know that each implicant contains a diagnosis. The theorem 1 states that any DNF description of the observed system contains the set of minimal diagnoses. Now, suppose that we monitor and diagnose a distributed system by the means of a distributed diagnostic architecture made up of local diagnostic engines which gradually compute local implicants from the monitored subsystems. A consistent composition of local implicants from each diagnostic engine is actually an implicant for the global system. We note that, as soon as each diagnostic engine returns its first implicant, the composition task can start. In the next section we precise the notion of distributed system which differs from the usual notion of system in diagnosis by taking into account the shared and local acquaintance of each subsystem. We take advantage of this characterization for forgetting symbols and optimizing the composition task.

3 Diagnosing Peer-to-Peer Settings

We formalize our distributed model based diagnosis framework by means of Peer-to-Peer Inference Systems (P2PIS) proposed by [9], and extended in [1] for distributed reasoning. In a P2PIS, a so-called "inference peer" has only a partial knowledge of the global network (generally restricted to its acquaintance) and may have local information, only known by itself. In our work, an inference peer will for instance model the expected behavior of a real peer, a web service, or a subcircuit, of a distributed system. Let us denote by T the description of the global observed system. T is the (virtual) conjunction of all local theories T_p of peers p. Of course, in our framework, T will never be explicitly gathered and privacy of local knowledge will be ensured. T is built on the global variables vocabulary V (excluding mode variables), which can be partitioned into shared variables Sh and local variables Loc

- $Sh = \{v | \exists p \neq p' \text{s.t. } v \text{ appears both in } T_p \text{ and in } T_{p'}\}$
- $Loc = V \setminus Sh$

In addition to this partition, we have to add mode variables in order to be able to diagnose the system. We denote by F the set of all mode variables of

the system. Obviously, in order to build a global diagnosis, exchange of mode variables between peers has to be possible. Thus, the network will allow formulas built on variables from $Sh \cup F$ to be sent from peer to peer. We denote by V_p, Sh_p, Loc_p, F_p the vocabulary, the shared variables, the local variables and the mode variables symbols of any inference peer p.

3.1 A Network of DNF Models

In the previous section, we assumed that we were able to work directly on the DNF of T. Because T here is a conjunction of formulas, we may push this hypothesis to all T_p. If the first hypothesis, i.e. in the centralized case, may not be considered as a realistic one, at the opposite small peers will admit relatively small DNF encoding, and thus the second hypothesis, i.e. in the distributed case, is of practical interest. If all T_p are in DNF, then writing T in DNF can be done by the distribution property of $\wedge$ over $\vee$. More formally, we use the following operator for this purpose:

Definition 2 (Distribution ($\otimes$))
$T_1^\vee \otimes T_2^\vee = \{I_1 \wedge I_2 | I_1 \in T_1^\vee, I_2 \in T_2^\vee, I_1 \wedge I_2 \not\models \bot\}$

One may notice that inconsistent products are deleted, and, if the result is minimized, then this operator is exactly the *clause-distribution* operator of [16], but applied to DNF and products.

Because of privacy, and for efficiency purpose, let us introduce the following lemma stating that instead of distributing all theories before restricting the result to mode variables, one may first restrict all theories to shared and mode variables without loss.

Lemma 2. *Let $T^\vee$ be a description of an observed P2P system, F a consistent set of negative mode literals:*

$$(\otimes T_p^\vee)|_{\{Sh,F\}} = \otimes(T_p^\vee|_{\{Sh_p,F_p\}})$$

Sketch of Proof. Let I (resp. I') an implicant of $T_p^\vee$, (resp. $T_{p'}^\vee$). Local symbols from I do not appear in I', thus inconsistencies between I and I' can only come from shared symbols.

With this lemma and the first theorem we can show that minimal diagnoses can be computed with shared and mode literals only.

Theorem 2. *Let T be a description of an observed P2P system, F a consistent set of negative mode literals:*

$$min_\subseteq(Diag(T)) = min_\subseteq((\otimes(T_p^\vee|_{\{Sh_p,F_p\}}))|_{\{F\}})$$

Sketch of Proof. Let $T^\vee$ be a DNF global description of the observed system s.t. $T^\vee \equiv T$. By theorem 1 we know that $min_\subseteq(Diag(T)) = min_\subseteq(T^\vee|_{\{F\}})$. We have $T^\vee|_{\{F\}} = T^\vee|_{\{Sh,F\}}|_{\{F\}}$ since the restriction of $T^\vee$ on shared and faulty symbols does not delete any faulty symbol. Moreover, because $T^\vee \equiv \otimes T_p^\vee$, we deduce by the lemma 2 that $min_\subseteq(Diag(T)) = min_\subseteq((\otimes T_p^\vee|_{\{(Sh_p,F_p)\}})|_{\{F\}})$.

3.2 Distributions with Trees

We now focus on the distribution of consistent diagnoses between diagnostic engines. Here we consider that any peer may be able to initiate a diagnosis and may ask its neighborhood to help him for this task. When receiving a request for a diagnosis by an initiator, a peer will also query its acquaintances, according to its observation values, and will begin to answer to its initiator as soon as possible. Thus, the initial request will flood into the network top-down and answers will converge to the initial peer with a bottom-up traversal of the network. Implicitly, for a given request for a diagnosis, any peer will maintain who was its local initiator, and thus an implicit tree will be built in the network for each request.

We use this tree to efficiently compute the distribution of peers theories. Let us denote by A_p the subtree rooted in p and by $child(A_p, A_{p'})$ the relation between $A_{p'}$ and A_p s.t. $A_{p'}$ is a subtree of A_p. We note by Sh_{A_p} the variables shared by A_p and any other peer in the distributed system. We note T^{A_p} the theory defined as the conjunction of the theories of all peers occurring in the the subtree rooted in p.

$$T^{A_p} = \begin{cases} T_p^{\vee}|_{\{F_p, Sh_p\}}, \text{ if } \nexists p' \text{s.t.} child(A_p, A_{p'}) \text{ is set.} \\ (T_p^{\vee}|_{\{F_p, Sh_p\}} \otimes \bigotimes_{\{A_{p'} | child(A_p, A_{p'})\}} T^{A_{p'}})|_{\{Sh_{A_p}, F_{A_p}\}}, \text{otherwise} \end{cases}$$

The next theorem shows that we can compute global diagnoses by gradually forgetting shared acquaintances which correspond to local acquaintances of a subtree.

Theorem 3. *Let T be the global description of an observed system, $child(A_p, A_{p'})$ a relation defining a Tree on T rooted in r. then:*

$$min_{\subseteq}(Diag(T)) = min_{\subseteq}(T^{A_r})$$

Sketch of Proof. We use the theorem 2 and inductively prove that $\forall p, T^{A_p} = (\bigotimes_{q \in A_p} T_q^{\vee}|_{\{Sh_q, F_q\}})|_{\{Sh_{A_p}, F_{A_p}\}}$. Concerning the root r, we note that $Sh_{A_r}=\emptyset$, consequently $min_{\subseteq}(T^{A_r})$ only contains the set of minimal diagnoses.

Thus, intuitively, as soon as we know that a given variable cannot imply any inconsistency in other parts of the tree, we remove it. As answers will go back to the root, peers will filter out useless variables, and, hopefully, will reduce the number and the size of possible answers.

4 Algorithm

In this section, we present our message-passing algorithm M2DT, standing for "Minimal Diagnoses by Distributed Trees" (see algorithm 1). We call *neighbor* of p a peer that shares variables with p. As previously, A stands for the distributed cover tree, dynamically built by the algorithm. We write A_p the subtree of A rooted in p. For a tree A and a peer p, p's *parent* and p's *children* will be included, by construction, in p's neighborhood. Let us recall that T^{A_p} is defined

as the theory (more exactly a subpart of the whole theory, sufficient for diagnostic purpose) of the observed subsystem defined by the conjunction of all peers occurring in the whole subtree A_p. We call *r-implicant* of T^{A_p} a restriction of one implicant of T^{A_p} to its mode variables and shared vocabulary.

4.1 A General View on M2DT

At the beginning, a given peer, called the *starter*, broadcasts a request of diagnosis (*reqDiag*) to its neighborhood. When a peer receives its first *reqDiag*, it sets the sender as its parent and broadcasts the request to its remaining neighbors, in order to flood the network. This first stage of the algorithm aims at building a distributed cover tree: as the request goes along the network, the relationship (*parent*, p) is set and defines the distributed cover tree A. As soon as one peer knows that it is a leaf in A, it answers by sending its r-implicants (*respDiag*) to its parent and thus begins the second stage of the algorithm. When an intermediate node receives r-implicants from one of its children, there are two cases. If it already knows the role of all its neighborhood (parent, direct children and peers that can either occur deeper in the current subtree or elsewhere in the cover tree), it extends all new r-implicants by distributing them over its own r-implicants and those already received from all other children. It then filters out useless variables and sends all resulting implicants to its parent. If it does not know the role of all its neighborhood, it stores the received r-implicants for a future use. With this algorithm, global diagnoses converge to the starter peer. When a peer does not wait any more for any message, it sends its termination message to its parent. When a peer has received all termination messages from all its children, it sends its termination message to its parent (third and last stage of the algorithm). When the starter peer receives the termination message, we are sure that it already received the set of minimal diagnoses from all its children.

4.2 Structures and Algorithm

A message can be a request of diagnosis *reqDiag*, a response *respDiag* or a notification of termination *endDiag*. The structure of a message *msg* is the following one:

- $msg.Type$: takes its values in {*reqDiag*, *respDiag*, *endDiag*}, matching the three stages of the algorithm.
- $msg.Desc$: defined only when $msg.Type = respDiag$, represents the descendants of the sender of the message that participated in building the considered r-implicant.
- $msg.rImpl$: defined only when $msg.Type = respDiag$, is an r-implicant of the subtree rooted in the sender of the message.

A peer p sets its *parent* to the first peer in its neighborhood that sent it a *reqDiag* message. For all other *reqDiag* messages that p may receive, it adds the sender to

the set $NotChild$. This set stores all peers that are not direct children of p (peers that can occur deeper in the subtree rooted in p or that do not occur in this subtree). All peers p' that send to p at least one $respDiag$ message are stored in $Child$. The array $TChild$, defined in each peer p only for its direct children p', associates to each child peer p' a DNF theory $TChild[p']$. $TChild[p']$ stores all r-implicants received so far from p'. This set will be known to be complete when p will receive an $endDiag$ message from p'. In order to detect additional useless variables, p also stores in $Desc$ all known descendant of p (peers occurring in the subtree rooted in p). All local variables of all peers are already deleted by the algorithm, but one may now consider as "local" a variable that is guaranteed to occur only in the current subtree and not elsewhere. This is the case for shared variables that are shared only by peers that occur in the current subtree.

To detect and notify termination, p maintains a list of peers from which messages are still waited. This list is called $waitEnd$ and initially set to all neighbors of p ($Neighborhood$). A peer leaves the list if it is the parent, if it is not a direct descendant or if it is a direct child that notified termination.

4.3 Primitives of M2DT

checkEnd $(waitEnd, p')$ Checks and propagates the termination. First, it removes p' from the $waitEnd$ list of p. If $waitEnd$ is empty, it sends the termination message and terminates.

extends $(I, T_p^\vee, TChild, Desc)$ Extends the implicant I from p' by distributing it on the local theory $T_p^\vee$ and all sets $TChild[p"]$ that are defined and different from p'. This primitive, which is only called when the local subtree is entirely known, computes
$(T_p^\vee|_{\{F_p, Sh_p\}} \otimes I \underset{p" \neq p'}{\otimes} TChild[p"])|_{\{Sh_{A_p}, F_{A_p}\}}$
One may notice that Sh_{A_p} is not directly known. It is deduced from $Desc$: we associate in $Desc$ to each shared variable the unique identifiers of all peers that share it. Thus, one may check if all peers that share a given shared variable are "local" to the subtree, only with the help of the set $Desc$.

flush $(T_p^\vee, TChild, Desc)$ Sends the distribution of all implicants stored in the $TChild$ array and the local theory. This primitive is called only when the local subtree is known to be complete for the first time with a $reqDiag$ message, which means that the last unknown neighbor sent to p a message "I am not your direct child". We thus have to flush all previously stored implicants (if any) to p's parent. This primitive computes
$(T_p^\vee|_{\{F_p, Sh_p\}} \underset{p" \neq p'}{\otimes} TChild[p"])|_{\{Sh_{A_p}, F_{A_p}\}}$

One may notice that this primitive will be called for all leaves of the distributed tree A.

4.4 Properties

In the following we assume a FIFO channel between two connected peers and no lost message. The acquaintance graph is connected and the global theory is

Algorithm 1. Peer p receives a message msg from peer p'

```
1: switch msg.Type
2:
3:  case : reqDiag
4:    /*A distributed tree is built*/
5:      if parent is not set then /* Flooding alg.*/
6:          parent ← p'
7:          send to all p neighborhood \p' : msg [reqDiag]
8:      else /* p' is not a direct child */
9:          NotChild ← NotChild ∪ {p'}
10:     end if
11:     /* Flushes all stored implicants when the subtree is known */
12:     if {parent} ∪ Child ∪ NotChild = Neighborhood
13:         Π ← flush(T_p^∨, TChild, Desc)
14:         for all I ∈ Π
15:           send to parent msg [respDiag,I,Desc ∪ {p}]
16:         end for
17:     end if
18:     /* p' is either the parent or not a direct child*/
19:     checkEnd(waitEnd, p')
20:
21:  case : respDiag
22:     /* Stores the diag, or extends and propagates it */
23:     Child ← Child ∪ {p'}
24:     Desc ← Desc ∪ msg.Desc
25:     TChild[p'] ← TChild[p'] ∪ msg.rImpl
26:     /* Extends msg.rImpl only if the subtree is already known */
27:     if {parent} ∪ Child ∪ NotChild = Neighborhood
28:         Π ← extends(msg.rImpl, T_p^∨, TChild, Desc)
29:         for all I ∈ Π
30:           send to parent msg [respDiag,I,Desc ∪ {p}]
31:         end for
32:         Tresult ← min_⊆(Tresult ∪ Π)
33:     end if
34:
35:  case : endDiag
36:  /* Notifies termination of this child, and propagates if needed*/
37:     checkEnd(waitEnd, p')
38: end switch
```

satisfiable. Messages processing is considered as "atomic", which simply means that the messages are treated one by one by each peer.

Let us first emphasize some observations.

Lemma 3. *If a peer, p, sends a reqDiag to one of its neighbors, p', this message is the only one from p to p'.*

Proof. A peer broadcasts the *reqDiag* to each of its neighbors (except its parent) just after having set as its parent the sender of the first received *reqDiag*. Because

of the condition line 5, p does not have the opportunity to send other *reqDiag*. Concerning *respDiag* and *endDiag* they are sent to the parent only.

If we now focus on the first event carried out by each peer:

Lemma 4. *All peers, except the starter, will receive a first event, which will be a reqDiag message.*

Proof. To receive a *respDiag* or an *endDiag*, p has to be a *parent* of some peer. But to become a parent, p must send a *reqDiag* to at least one peer, and thus p will have to receive *reqDiag* first. Consequently, since the acquaintance graph is connected, and because a peer broadcasts *reqDiag* to its neighbors, then each peer will receive a first *reqDiag* (we rely on the well known flooding algorithm in a graph to ensure this).

With these lemmas we have the following property:

Property 1 (Distributed cover Tree). The relation (*parent*,p), built when p receives its first *reqDiag*, defines a distributed cover tree.

Sketch of Proof. Let n be the number of peers, $req_1(p)$ be the reception of the first diagnosis request by p. Since the starter peer does not get any parent, by the previous lemma we know that flooding *reqDiag* will build n-1 connections (parent,p). Suppose a cycle is defined by these connections and an order $<$ s.t. $req_1(p) < req_1(p')$ if $req_1(p)$ is former than $req_1(p')$. Let us take p in the cycle, there exists p' in the cycle s.t. p' got p as parent, then $req_1(p) < req_1(p')$. If we follow the "parent" connection in the cycle, we have by transitivity that $req_1(p') < req_1(p)$. Consequently, we cannot have cycle by the "parent" connection.

Now we know that a distributed cover tree will be built but, at this point, a peer p will only know its parent, but not its direct children. This can be deduced by the *respDiag* messages, when all children of p will send it their r-implicants. *respDiag* messages are only sent when the state of all neighbors of p are known (Parent, Child, NotChild).

Lemma 5. *Let p be a peer, p' be one of its neighbors. If p' does not get p as parent, p will receive a reqDiag from p'.*

Sketch of Proof. Let p' be a neighbor of p that does not accept p as its parent. When p sent to it a *reqDiag*, p' already had a parent in order to refuse p as its parent. Consequently p had also sent to p' a *reqDiag*.

With this lemma, we can easily show the following property:

Property 2. Let A_p be the subtree rooted in p and built by the algorithm, T^{A_p}, the theory of A_p as previously defined, then p will send to its parent the r-implicants of T^{A_p}.

Sketch of Proof. Let us prove this property recursively, with respect to the maximal depth, d_{max}, of the subtree A_p, rooted in p. If d_{max}=0, p is a leaf, none of its neighbors gets it as parent. Nevertheless, with the previous lemma, we know that each of them will send a *reqDiag* to it. Consequently, p will know the state of its

neighbors and satisfy the condition ($parent \cup Child \cup NotChild = Neighborhood$). Then p will send all r-implicants of $T_p^{\vee}$ computed by the flush primitive. If $d_{max} > 0$, suppose the property true for all children p' of p: we thus guarantee that p will receive from each of them their r-implicants and will store them in its $TChild$ array. Concerning the other neighbors (NotChild), p will receive from them a $reqDiag$. Consequently, p will know the state of all its neighborhood and satisfy the condition ($parent \cup Child \cup NotChild = Neighborhood$). Since the global theory is satisfiable p will be able to build at least one r-implicant of T^{A_p} either by the method extends or by the method flush.

Since $TResult$ is minimized, the starter peer will save in it the minimal diagnoses. The correction and the completeness of the algorithm are a direct consequence of the previous property. The termination of the algorithm is shown by the following one:

Property 3. The last event carried out by a peer is sending the *endDiag* message to its parent.

Sketch of Proof. Similarly to the previous property, we can show this property recursively with respect to the maximal depth, d_{max}. If $d_{max} = 0$, p will receive a *reqDiag* from all its neighbors and the set *waitEnd* will be empty. Then p will send the message *end* after the set of r-implicants of $T_p^{\vee}$. If $d_{max} > 0$, p will receive a *reqDiag* from peers in NotChild and an *endDiag* from its children. Then, the set *waitEnd* will be empty. As soon as possible, p will send r-implicants to its parent and an *endDiag* as its last message.

Example 5 (M2DT Illustration). A customer made a hire purchase by internet whereas his credit card was not valid: the service CCS observed $\neg valCC$ and the service ES observed hPurch. OVS starts the analysis and sends a diagnosis request to ES. ES begins the computation of $T_{ES}^{\vee}|_{\{Sh_{ES}, F_{ES}\}}$ and forwards the request to its neighbors HPS and LAS. HPS and LAS receive the diagnosis request from ES and both forward a diagnosis request to CCS. CCS forwards the request to LAS. When LAS receives the request from CCS it has already received one from ES, so it does not answer. At this step, LAS has received a message from all its neighborhood, then it starts to send its implicants to ES. When

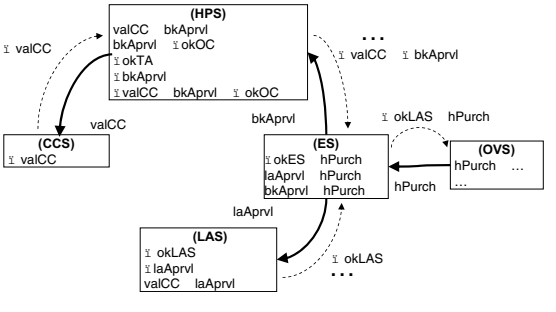


Fig. 2. M2DT algorithm

CCS receives the request from LAS, it does not answer and sends its implicant $\neg valCC$ *to HPS. Simultaneously, ES gets* $\neg laAprvl$ *from LAS and HPS gets* $\neg valCC$ *from CCS. At this step, ES did not received any message from any of its neighbors, unlike HPS, which can send to ES the implicant* $\neg bkAprvl \wedge \neg valCC$ *built from the 4th implicant of its theory and* $\neg valCC$*. At this step, ES received a message from its neighbors. It builds* $\neg okLAS \wedge \neg bkAprvl \wedge hPurch \wedge \neg valCC$ *from its theory and its received implicants. ES removes* $\neg bkAprvl$ *and* $\neg valCC$ *which are shared variables only occurring in the subtree rooted in ES and sends* $\neg okLAS \wedge hPurch$ *to OVS. At this step OVS gets its first diagnosis.*

5 Related Works

Our approach has been preceded by many pieces of work. Methods from [2] extend the ATMS principles of [7] in order to incrementally compute the set of conflicts in a distributed framework. However, those methods have still to wait for all conflicts before having a chance to get the first diagnosis. Many other works try to take advantage of the system topology, for instance by using decomposition properties of the model [5,13]. Similarly [12] searches for diagnoses into a partitioned graph by assigning values for shared variables and maintaining a local consistency. The global diagnosis is thus distributed in all local diagnoses. This method however does not guarantee the minimality and supposes a global system description, which is not our case. In [3], local diagnoses from each agent (peer) are synchronized such that to obtain at the end a compact representation of global diagnoses. But the algorithm searches only for the set of diagnoses with minimal cardinality whereas we look for the set of all minimal diagnoses for subset inclusion. In [15], agents update their sets of local diagnoses in order to be consistent with a global one. However, the algorithm cannot guarantee that any combination of agents local minimal diagnoses is also a global minimal diagnosis. In [4,10], a supervisor, that knows the global communication architecture between peers, coordinates the diagnosis task by dialoguing with local diagnosers. Thus the computation of global diagnoses from local ones is centralized. Due to the privacy constraint, in our framework a peer just knows its neighbors, no peer knows the network architecture and computation of global diagnoses is distributed, no peer playing a special role.

6 Conclusion

We proposed a distributed algorithm to compute the minimal diagnoses of a distributed Peer-to-Peer setting in an incremental way, with the help of a distributed cover tree of the acquaintance graph of the peers. Our algorithm takes advantage of the DNF representation of the local theories of the peers in order to compute global diagnoses without needs to get conflicts first. However, one has to notice that, in practice, peers do not have to rewrite their local theories in DNF. They may compute answers to requests "on the fly" and thus allow our algorithm to work on CNF encoding of the peers. Developing this technique will be our next

investigation, along with experimental testing and scaling-up study of our algorithm on real examples, such as conversational web services. We can already state the many advantages offered by our approach: we ensure privacy issues, in particular no peer has the global knowledge; we never compute the set of conflicts before computing the diagnoses; we take advantage of the natural structure of the network, which can be generally decomposed such as to obtain a small number of shared variables; we take advantage of the distributed cpu power of the whole network; lastly, we restrict the vocabulary of diagnoses as soon as possible. Future lines of research will include the study of dynamic Peer-to-Peer systems.

References

1. Adjiman, P., Chatalic, P., Rousset, M.-C., Simon, L.: Distributed reasoning in a peer-to-peer setting. In: IJCAI 2005 (2005)
2. Beckstein, C., Fuhge, R., Kraetzschmar, G.: Supporting assumption-based reasoning in a distributed environment. In: Workshop on Distributed Artificial Intelligence (1993)
3. Biteus, J., Frisk, E., Nyberg, M.: Distributed diagnosis by using a condensed local representation of the global diagnoses with minimal cardinality. In: DX 2006 (2006)
4. Console, L., Picardi, C., Theseider Dupré, D.: A framework for decentralized qualitative model-based diagnosis. In: IJCAI 2007 (2007)
5. Darwiche, A.: Model-based diagnosis using structured system descriptions. Journal of AI Research 8, 165–222 (1998)
6. Darwiche, A., Marquis, P.: A knowledge compilation map. Journal of AI Research 17, 229–264 (2002)
7. de Kleer, J.: An assumption-based tms. Artificial Intelligence 28, 127–162 (1986)
8. de Kleer, J., Mackworth, A.K., Reiter, R.: Characterizing diagnoses and sytems. Artificial Intelligence 56, 197–222 (1992)
9. Halevy, A., Ives, Z., Tatarinov, I.: Schema mediation in peer data management systems. In: ICDE 2003, pp. 505–516 (March 2003)
10. Kalech, M., Gal Kaminka, A.: On the design of social diagnosis algorithms for multi-agent teams. In: IJCAI 2003 (2003)
11. de Kleer, J., Williams, B.C.: Diagnosing multiple faults. Artificial Intelligence 32(1), 97–130 (1987)
12. Kurien, J., Koutsoukos, X., Zhao, F.: Distributed diagnosis of networked, embedded systems. In: DX 2002 (2002)
13. Provan, G.: A model-based diagnosis framework for distributed systems. In: DX 2002 (2002)
14. Reiter, R.: A theory of diagnosis from first principles. Artificial Intelligence 32(1), 57–96 (1987)
15. Roos, N., ten Teije, A., Witteveen, C.: A protocol for multi agent diagnosis with spatially distributed knowledge. In: AAMAS 2003 (2003)
16. Simon, L., del Val, A.: Efficient consequence finding. In: IJCAI 2001 (2001)
17. Stoica, I., Morris, R., Karger, D., Kaasshoek, M.F., Balakrishnan, H.: Chord: a scalable peer-to-peer lookup service for internet applications. In: ACM SIGCOMM 2001 (2001)

From One Session to Many: Dynamic Tags for Security Protocols*

Myrto Arapinis, Stéphanie Delaune, and Steve Kremer

LSV, ENS Cachan & CNRS & INRIA, France

Abstract. The design and verification of cryptographic protocols is a notoriously difficult task, even in abstract Dolev-Yao models. This is mainly due to several sources of unboundedness (size of messages, number of sessions, ...). In this paper, we characterize a class of protocols for which secrecy for an unbounded number of sessions is decidable. More precisely, we present a simple transformation which maps a protocol that is secure for a single protocol session (a decidable problem) to a protocol that is secure for an unbounded number of sessions.

Our result provides an effective strategy to design secure protocols: (i) design a protocol intended to be secure for one protocol session (this can be verified with existing automated tools); (ii) apply our transformation and obtain a protocol which is secure for an unbounded number of sessions. The proof of our result is closely tied to a particular constraint solving procedure by Comon-Lundh *et al.*

1 Introduction

Security protocols are small distributed programs which aim at guaranteeing properties such as confidentiality of data, authentication of participants, etc. The security of these protocols relies on the one hand on the security of cryptographic primitives, e.g. encryption and digital signatures, and on the other hand on the concurrency-related aspects of the protocols themselves. History has shown that even if cryptography is supposed to be perfect, such as in the classical Dolev-Yao model [16], the correct design of security protocols is notoriously error-prone. See for instance [7] for an early survey on attacks. These difficulties come mainly from two sources of unboundedness: a protocol may be executed several times (we get several protocol *sessions*) and the attacker is allowed to build messages of unbounded size. Indeed, secrecy is known to be undecidable when an unbounded number of sessions is allowed, even if the message size is bounded [17]. However, when the number of sessions is bounded, and even without assuming a bounded message size, the problem becomes co-NP-complete [25]. Moreover, special purpose verification tools (e.g. [3]) exist which are highly efficient when the number of sessions is small.

In this paper we consider the secrecy property and we propose a protocol transformation which maps a protocol that is secure for a single session to a protocol that is secure for an unbounded number of sessions. This provides an effective strategy to design secure protocols: (i) design a protocol intended to be secure for one protocol session

* Work partly supported by ARA SSIA Formacrypt and CNRS/JST ICT "Cryptography and logic: Computer-checked security proofs".

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 128–142, 2008.

(this can be efficiently verified with existing automated tools); (ii) apply our transformation and obtain a protocol which is secure for an unbounded number of sessions.

Our transformation. Suppose that Π is a protocol between k participants $A_1, \ldots, A_k$. Our transformation adds to Π a preamble in which each participant sends a freshly generated nonce N_i together with his identity to all other participants. This allows each participant to compute a dynamic, session dependent *tag* $\langle A_1, N_1 \rangle, \ldots, \langle A_k, N_k \rangle$ that will be used to tag each encryption and signature in Π. Our transformation is surprisingly simple and does not require any cryptographic protection of the preamble. Intuitively, the security relies on the fact that the participant A_i decides on a given tag for a given session which is ensured to be fresh as it contains his own freshly generated nonce N_i. The transformation is computationally light as it does not add any cryptographic application; it may merely increase the size of messages to be encrypted or signed. The transformation applies to a large class of protocols, which may use symmetric and asymmetric encryption, digital signature and hash function.

We may note that, *en passant*, we identify a class of tagged protocols for which security is decidable for an unbounded number of sessions. This directly follows from our main result as it stipulates that verifying security for a single protocol session is sufficient to conclude security for an unbounded number of sessions.

Related Work. The kind of compiler we propose here has also been investigated in the area of cryptographic design in computational models, especially for the design of group key exchange protocols. For example, Katz and Yung [19] proposed a compiler which transforms a key exchange protocol secure against a passive eavesdropper into an authenticated protocol which is secure against an active attacker. Earlier work includes compilers for 2-party protocols (e.g. [5]). In the symbolic model, recent works [12,4] allow one to transform a protocol which is secure in a weak sense (roughly no attacker [12] or just a passive one [4] and a single session) into a protocol secure in the presence of an active attacker and for an unbounded number of sessions. All of these works share however a common drawback: the proposed transformations make heavy use of cryptography. This is mainly due to the fact that the security assumptions made on the input protocol are rather weak. As already mentioned in [12], it is important, from an efficiency perspective to lighten the use of cryptographic primitives. In this work, we succeed in doing so at the price of requiring stronger security guarantees on the input protocol. However, we argue that this is acceptable since efficient automatic tools exist to decide this security criterion on the input protocols.

We can also compare our work with existing decidable protocol classes for an unbounded number of sessions. An early result is the PTIME complexity result by Dolev *et al.* [15] for a restricted class, called *ping-pong* protocols. Other classes have been proposed by Ramanujam and Suresh [23,24], and Lowe [21]. However, in both cases, temporary secrets, composed keys and ciphertext forwarding are not allowed which discards protocols, such as the Yahalom protocol [7] (see also Section 4.3).

Outline of the paper. In Section 2 we describe the term algebra which is used to model protocol messages as well as the intruder capabilities to manipulate such terms. Then, in Section 3, we define the protocol language we use to model protocols. In Section 4

we formally describe our transformation and state our main transference result. Finally, in Section 5, we prove our main result. Due to a lack of space the proofs are given in [1].

2 Messages and Intruder Capabilities

2.1 Messages

We use an abstract term algebra to model the messages of a protocol. For this we fix several disjoint sets. We consider an infinite set of *agents* $\mathcal{A} = \{\epsilon, a, b \ldots\}$ with the special agent ϵ standing for the attacker and an infinite set of *agent variables* $\mathcal{X} = \{x_A, x_B, \ldots\}$. We need also to consider an infinite set of *names* $\mathcal{N} = \{n, m \ldots\}$ and an infinite set of *variables* $\mathcal{Y} = \{y, z, \ldots\}$. We consider the following *signature* $\mathcal{F} = \{\mathsf{enc}/2, \mathsf{enca}/2, \mathsf{sign}/2, \langle\,\rangle/2, \mathsf{h}/1, \mathsf{priv}/1, \mathsf{shk}/2\}$. These function symbols model cryptographic primitives. The symbol $\langle\,\rangle$ represents pairing. The term $\mathsf{enc}(m, k)$ (resp. $\mathsf{enca}(m, k)$) represents the message m encrypted with the symmetric (resp. asymmetric) key k whereas the term $\mathsf{sign}(m, k)$ represents the message m signed by the key k. The function h models a hash function whereas $\mathsf{priv}(a)$ is used to model the private key of an agent a, and $\mathsf{shk}(a, b)$ (= $\mathsf{shk}(b, a)$) is used to model the long-term symmetric key shared by agents a and b. For simplicity, we confuse the agent names with their public key. Names are used to model atomic data such as nonces. The set of *terms* is defined inductively by the following grammar:

$$t, t_1, t_2 \ldots ::= y \mid n \mid x \mid a \mid \mathsf{priv}(u_1) \mid \mathsf{shk}(u_1, u_2) \mid f(t_1, t_2) \mid \mathsf{h}(t)$$

where $u_1, u_2 \in \mathcal{A} \cup \mathcal{X}$, and $f \in \{\langle\,\rangle, \mathsf{enc}, \mathsf{enca}, \mathsf{sign}\}$. We sometimes write $\langle t_1, \ldots, t_n \rangle$ instead of writing $\langle t_1, \langle, \ldots, \langle t_{n-1}, t_n \rangle \ldots \rangle \rangle$. We say that a term is *ground* if it has no variable. We consider the usual notations for manipulating terms. We write $vars(t)$ (resp. $fresh(t)$, $agent(t)$) for the set of variables (resp. names, agents) occurring in t. We write $St(t)$ for the set of *subterms* of a term t and define the set of *cryptographic subterms* of a term t as $CryptSt(t) = \{f(t_1, \ldots, t_n) \in St(t) \mid f \in \{\mathsf{enc}, \mathsf{enca}, \mathsf{sign}, \mathsf{h}\}\}$. Moreover we define the set of *long-term keys* of a term t as

$$lgKeys(t) = \{\mathsf{priv}(u) \mid u \in \mathcal{A} \cup \mathcal{X} \text{ and } u \in St(t)\} \cup \{\mathsf{shk}(u_1, u_2) \in St(t)\}.$$

and we define $\mathcal{K}_\epsilon = \{\mathsf{priv}(\epsilon)\} \cup \{\mathsf{shk}(a, \epsilon) \mid a \in \mathcal{A}\}$. Intuitively $\mathcal{K}_\epsilon$ represents the set of long-term keys of the attacker. An *atom* is a long-term key, a name or a variable. We define the set of *plaintexts* of a term t as the set of atoms that occur in plaintext, i.e

- $plaintext(\mathsf{h}(u)) = plaintext(f(u, v)) = plaintext(u)$ for $f \in \{\mathsf{enc}, \mathsf{enca}, \mathsf{sign}\}$,
- $plaintext(\langle u, v \rangle) = plaintext(u) \cup plaintext(v)$, and
- $plaintext(u) = \{u\}$ otherwise.

All these notions are extended to sets of terms and to other kinds of term containers as expected. We denote by $\#S$ the cardinality of a set S. Substitutions are written $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ where its *domain* is $\mathrm{dom}(\sigma) = \{x_1, \ldots, x_n\}$. The substitution σ is *ground* if all the t_i are ground. The application of a substitution σ to a term t is written $\sigma(t)$ or $t\sigma$. Two terms t_1 and t_2 are *unifiable* if $t_1\sigma = t_2\sigma$ for some substitution σ, that is called a *unifier*. We denote by $\mathrm{mgu}(t_1, t_2)$ the *most general unifier* of t_1 and t_2.

$$\frac{T \vdash u \quad T \vdash v}{T \vdash \langle u, v \rangle} \qquad \frac{T \vdash u \quad T \vdash v}{T \vdash \mathsf{enc}(u, v)} \qquad \frac{T \vdash u \quad T \vdash v}{T \vdash \mathsf{enca}(u, v)} \qquad \frac{T \vdash u \quad T \vdash v}{T \vdash \mathsf{sign}(u, v)} \qquad \frac{T \vdash u}{T \vdash \mathsf{h}(u)}$$

$$\frac{T \vdash \langle u, v \rangle}{T \vdash u} \qquad \frac{T \vdash \langle u, v \rangle}{T \vdash v} \qquad \frac{T \vdash \mathsf{enc}(u, v) \quad T \vdash v}{T \vdash u}$$

$$\frac{T \vdash \mathsf{enca}(u, v) \quad T \vdash \mathsf{priv}(v)}{T \vdash u} \qquad \frac{T \vdash \mathsf{sign}(u, v)}{T \vdash u} \textit{ (optional)} \qquad \frac{}{T \vdash u}\, u \in T \cup \mathcal{A} \cup \mathcal{K}_\epsilon$$

Fig. 1. Intruder deduction system

Example 1. Let $t = \mathsf{enc}(\langle n, a \rangle, \mathsf{shk}(a, b))$. We have that $vars(t) = \emptyset$, i.e. t is ground, $fresh(t) = \{n\}$, $agent(t) = \{a, b\}$, $lgKeys(t) = \{\mathsf{priv}(a), \mathsf{priv}(b), \mathsf{shk}(a, b)\}$ and $plaintext(t) = \{n, a\}$. The terms $\mathsf{shk}(a, b)$, a, n and $\mathsf{priv}(a)$ are atoms.

2.2 Intruder Capabilities

We model the intruder's abilities to construct new messages by the deduction system given in Figure 1. The intuitive meaning of these rules is that an intruder can compose new messages by pairing, encrypting, signing and hashing previously known messages provided he has the corresponding keys. Conversely, he can decompose messages by projecting or decrypting provided he has the decryption keys. Our optional rule expresses that an intruder can retrieve the whole message from its signature. Whether this property holds depends on the actual signature scheme. Therefore we consider this rule to be optional. Our results work in both cases.

Definition 1 (Deducible). *We say that a term u is* deducible *from a set of terms T, denoted $T \vdash u$, if there exists a tree such that its root is labeled by $T \vdash u$ and for every node labeled by $T \vdash v$ having n sons labeled by $T \vdash v_1, \ldots, T \vdash v_n$ we have that $\frac{T \vdash v_1, \ldots, T \vdash v_n}{T \vdash v}$ is an instance of one of the inference rules given in Figure 1.*

Example 2. The term $\langle n, \mathsf{shk}(a, b) \rangle$ is deducible from $\{\mathsf{enc}(n, \mathsf{shk}(a, b)), \mathsf{shk}(a, b)\}$.

3 Model for Security Protocols

In this section, we give a language for specifying protocols and define their execution in the presence of an active attacker. Our model is similar to existing ones (see *e.g.* [25, 10]) and mostly inspired by [11] except for the fact that we introduce *parametrized roles* which allows us to instantiate new sessions.

3.1 Syntax

Our protocol model allows parties to exchange messages built from identities and randomly generated nonces using public key and symmetric encryption, digital signature and hashing. The individual behavior of each protocol participant is defined by a *role*. A role describes a sequence of *events*, i.e. a sequence of receiving and sending.

Definition 2 (Event, role and protocol). *An* event e *is either a* receive event*, denoted* $\mathsf{rcv}(u)$*, or a* send event*, denoted* $\mathsf{snd}(u)$*, where* u *is a term. A* role *is of the form* $\lambda x_1 \ldots \lambda x_k . \nu y_1 \ldots \nu y_p .\ \mathsf{seq}$*, where*

- $X = \{x_1, \ldots, x_k\}$ *is a set of agent variables, i.e. the parameters of the role corresponding to the* k *participants of the protocol,*
- $Y = \{y_1, \ldots, y_p\}$ *is a set of variables: the nonces generated by the role,*
- $\mathsf{seq} = \mathsf{e}_1; \mathsf{e}_2; \ldots; \mathsf{e}_\ell$ *is a sequence of events such that* $(vars(\mathsf{seq}) \setminus \{X\}) \subseteq \mathcal{Y}$*, i.e. all agents variables are parameters.*

Moreover, for all $i,\ 1 \leq i \leq \ell$*, we have that* $\mathsf{e}_i = \mathsf{snd}(u)$ *implies*

1. $vars(u) \subseteq X \cup Y \cup \{vars(v) \mid \mathsf{e}_j = \mathsf{rcv}(v)$ *and* $j < i\}$*, and*
2. $vars(plaintext(u)) \subseteq X \cup Y \cup \{vars(plaintext(v)) \mid \mathsf{e}_j = \mathsf{rcv}(v)$ *and* $j < i\}$.

The set of roles is denoted by Roles*. The* length *of a role is the number of elements in its sequence of events. A* k-party protocol *is a mapping* $\Pi : [k] \to \mathsf{Roles}$*, where* $[k] = \{1, 2, \ldots, k\}$.

The two last conditions on variables only discard protocols that are not executable by requiring that each variable which appears in a sent term (at a plaintext position) is either one of the parameters, nonces, or is a variable which has been bound by a previous receive event (at a plaintext position). Condition 1 is rather standard whereas Condition 2 will be useful later on to trace data that occur in plaintext position.

Example 3. We illustrate our protocol syntax on the familiar Needham-Schroeder public-key protocol. In our syntax this protocol is modeled as follows.

$$\begin{aligned}\Pi(1) = \lambda x_A.\lambda x_B.\nu y.\ & \mathsf{snd}(\mathsf{enca}(\langle y, x_A\rangle, x_B));\\ & \mathsf{rcv}(\mathsf{enca}(\langle y, z\rangle, x_A));\\ & \mathsf{snd}(\mathsf{enca}(z, x_B))\end{aligned} \qquad \begin{aligned}\Pi(2) = \lambda x_A.\lambda x_B.\nu y'.\ & \mathsf{rcv}(\mathsf{enca}(\langle z', x_A\rangle, x_B));\\ & \mathsf{snd}(\mathsf{enca}(\langle z', y'\rangle, x_A));\\ & \mathsf{rcv}(\mathsf{enca}(y', x_B))\end{aligned}$$

The initiator, role $\Pi(1)$ played by x_A, sends to the responder, role $\Pi(2)$ played by x_B, his identity together with a freshly genrated nonce y, encrypted with the responder's public key. The reponder replies by copying the initiator's nonce and adds a fresh nonce y', encrypted by the initiator's public key. The initiator acknowledges by forwarding the responder's nonce encrypted by its public key.

3.2 Scenarios and Sessions

In our model, a session corresponds to the instantiation of one role. This means in particular that one "normal execution" of a k-party protocol requires k sessions, one per role[1]. We may want to consider several sessions corresponding to different instantiations of a same role. Since the adversary may block, redirect and send new messages, all the sessions might be interleaved in many ways. Such an interleaving is captured by the notion of a *scenario*.

[1] In the literature, the word session is often used in an abusive way to represent an execution of the *protocol*, i.e. one session per role, whereas we use it for the execution of a *role*.

Definition 3 (Scenario). *A* scenario for a protocol $\Pi : [k] \to \mathsf{Roles}$ *is a sequence* $\mathsf{sc} = (r_1, sid_1) \cdots (r_n, sid_n)$ *where* r_i *is a role and* sid_i *a session identifier such that* $1 \leq r_i \leq k$, $sid_i \in \mathbb{N} \setminus \{0\}$, *the number of identical occurrences of a pair* (r, sid) *is smaller than the length of the role* r, *and* $sid_i = sid_j$ *implies* $r_i = r_j$.

The condition on identical occurrences ensures that a role cannot execute more events than it contains. The last condition ensures that a session number is not reused on other roles. We say that $(r, s) \in \mathsf{sc}$ if (r, s) is an element of the sequence sc.

Given a scenario and an instantiation for the parameters, we define a *symbolic trace*, that is a sequence of events that corresponds to the interleaving of the scenario, for which the parameters have been instantiated, fresh nonces are generated and variables are renamed to avoid name collisions between different sessions.

Definition 4 (Symbolic trace). *Let* Π *be a* k*-party protocol with*

$$\Pi(j) = \lambda x_1^j. \ldots \lambda x_k^j. \nu y_1^j. \ldots \nu y_{p_j}^j. \mathsf{e}_1^j; \ldots; \mathsf{e}_{\ell_j}^j \quad \text{for } 1 \leq j \leq k.$$

Given a scenario $\mathsf{sc} = (r_1, sid_1) \cdots (r_n, sid_n)$ *and a function* $\alpha : \mathbb{N} \to \mathcal{A}^k$, *the* symbolic trace $\mathsf{tr} = \mathsf{e}_1; \ldots; \mathsf{e}_n$ associated to sc and α *is defined as follows. Let* $q_i = \#\{(r_j, sid_j) \in \mathsf{sc} \mid j \leq i, sid_j = sid_i\}$, *i.e. the number of previous occurrences in* sc *of the session* sid_i. *We have* $q_i \leq \ell_{r_i}$ *and* $\mathsf{e}_i = (\mathsf{e}_{q_i}^{r_i})\sigma_{r_i, sid_i}$, *where*

- $\mathrm{dom}(\sigma_{r,sid}) = \{x_1^r, \ldots, x_k^r\} \cup \{y_1^r, \ldots, y_{p_r}^r\} \cup vars(\Pi(r))$,
- $\sigma_{r,sid}(y) = n_{y,sid}$ *if* $y \in \{y_1^r, \ldots, y_{p_r}^r\}$, *where* $n_{y,sid}$ *is a fresh name;*
- $\sigma_{r,sid}(x_i^r) = a_i$ *when* $\alpha(sid) = (a_1, \ldots, a_k)$;
- $\sigma_{r,sid}(z) = z_{sid}$ *otherwise, where* z_{sid} *is a fresh variable.*

A session sid *is said to be* honest *w.r.t.* α *when* $\alpha(sid) \in (\mathcal{A} \setminus \{\epsilon\})^k$.

Intuitively, a session sid is honest if all of its participants, from the point of view of the agent playing the session sid, are honest (i.e. $\neq \epsilon$). Since agent variables only occur as parameters in a protocol (see Def. 2), a symbolic trace does not contain agent variables.

We define an operator K which associates to a symbolic trace tr the knowledge gained by the adversary, i.e. the set of (possibly non ground) terms that are sent in this symbolic trace. More precisely, we have that $\mathsf{K}(\mathsf{e}_1; \cdots; \mathsf{e}_\ell) = \bigcup_{1 \leq i \leq \ell} \mathsf{K}(\mathsf{e}_i)$ where $\mathsf{K}(\mathsf{rcv}(u)) = \emptyset$ and $\mathsf{K}(\mathsf{snd}(u)) = \{u\}$. This operator is useful in the following when we associate a constraint system to a symbolic trace.

3.3 Constraint Systems

Constraint systems have been successfully used for verifying secrecy properties of finite scenarios (see for instance [25,22,13]). We now recall the definition of constraint systems. In the next section we discuss how secrecy for an *unbounded number of sessions* can be specified using (infinite) families of constraint systems.

Definition 5 (Constraint system). *A* constraint system C *is either* $\bot$ *or a finite sequence of expressions* $(T_i \Vdash u_i)_{1 \leq i \leq n}$, *called* constraints. *Each* T_i *is a finite set of terms, called the* left-hand side *of the constraint, and each* u_i *is a term, called the* right-hand side *of the constraint. Moreover, we assume that terms in* C *do not contain agent variables and are such that:*

1. $T_i \subseteq T_{i+1}$ *for every* i *such that* $1 \leq i < n$;
2. *if* $x \in vars(T_i)$ *for some* $1 \leq i \leq n$ *then* $\exists j < i$ *such that* $x \in vars(u_j)$.

A solution *of* C *is a ground substitution* θ *with* $\mathrm{dom}(\theta) = vars(C)$ *such that for every* $(T \Vdash u) \in C$, *we have that* $T\theta \vdash u\theta$. *The empty constraint system is always satisfiable whereas* $\bot$ *denotes an unsatisfiable system. We denote by* $\mathsf{maxlhs}(C)$ *(resp.* $\mathsf{minlhs}(C)$*) the maximal (resp. minimal) left-hand side of* C, *i.e.* T_n *(resp.* T_1*). We denote by* $\mathsf{rhs}(C)$ *the set of its right-hand sides, i.e.* $\{u_1, \ldots, u_n\}$.

In the remainder of the paper we often consider constraint systems as sets rather than sequences of constraints, keeping the ordering induced by set inclusion of the left-hand side of constraints implicit. The left-hand side of a constraint system usually represents the messages sent on the network. Hence, the first condition states that the intruder knowledge is always increasing. The second condition in Definition 5 says that each variable occurs first in some right-hand side.

3.4 Secrecy

We now define the secrecy preservation problem for an unbounded number of sessions. Intuitively, a term m is secret if for all possible instantiations and scenarios, the ground term m' obtained when all parameters and nonces have been instantiated during an *honest* session remains secret. This definition leads us to consider an infinite family of constraint systems.

Definition 6 (Secrecy). *Let* Π *be a* k*-party protocol with*

$$\Pi(j) = \lambda x_1^j \ldots \lambda x_k^j . \nu y_1^j \ldots \nu y_{p_j}^j . \mathsf{seq}^j \quad \text{for } 1 \leq j \leq k.$$

and let $m \in St(\mathsf{seq}^r)$ *for some role* $1 \leq r \leq k$. *We say that* Π *preserves the secrecy of* m *w.r.t.* T_0 *(a finite set of ground atoms), if for any scenario* sc, *for any function* $\alpha : \mathbb{N} \to \mathcal{A}^k$ *and for any honest session* sid_h (i.e. $\alpha(sid_h) \in (\mathcal{A} \setminus \{\epsilon\})^k$) *such that* $(r, sid_h) \in \mathsf{sc}$, *the following constraint system is not satisfiable*

$$\{T_0 \cup \mathsf{K}(\mathsf{tr}_i) \Vdash u \mid \mathsf{tr}_i = \mathsf{tr}_{i-1}; \mathsf{rcv}(u) \text{ and } 1 \leq i \leq \ell\} \cup \{T_0 \cup \mathsf{K}(\mathsf{tr}) \Vdash m\sigma_{r,sid_h}\}$$

where tr *is the symbolic trace of length* ℓ *associated to* (sc, α) *and* tr_i *its prefix of length* i. *The substitution* σ_{r,sid_h} *is defined as in Definition 4.*

Example 4. Consider again the Needham-Schroeder protocol. Let $\Pi(1)$ and $\Pi(2)$ be the two roles introduced in Example 3. This protocol is well-known to be insecure w.r.t. $m = y'$ for any T_0. Let sid_1 and sid_2 be two session identifiers such that $sid_1 \neq sid_2$ and consider the scenario $\mathsf{sc} = (1, sid_1)\,(2, sid_2)\,(2, sid_2),\ (1, sid_1)\,(1, sid_1)$ and the function α such that $\alpha(sid_1) = (a, \epsilon)$ and $\alpha(sid_2) = (a, b)$. The constraint system C associated to T_0, sc, α and $m\sigma_{2,sid_2} = n_{y',sid_2}$ (according to Definition 6) is given below.

$$C := \left\{ \begin{array}{c} T_1 \stackrel{\mathsf{def}}{=} T_0,\ \mathsf{enca}(\langle n_{y,sid_1}, a\rangle, \epsilon) \Vdash \mathsf{enca}(\langle z'_{sid_2}, a\rangle, b) \\ T_2 \stackrel{\mathsf{def}}{=} T_1,\ \mathsf{enca}(\langle z'_{sid_2}, n_{y',sid_2}\rangle, a) \Vdash \mathsf{enca}(\langle n_{y,sid_1}, z_{sid_1}\rangle, a) \\ T_2,\ \mathsf{enca}(z_{sid_1}, \epsilon) \Vdash n_{y',sid_2} \end{array} \right.$$

The substitution $\sigma = \{z'_{sid_2} \mapsto n_{y,sid_1}, z_{sid_1} \mapsto n_{y',sid_2}\}$ is a solution of C. However, this protocol preserves the secrecy of m (w.r.t. $T_0 = \emptyset$ for instance) when considering one honest session for each role. This has been formally verified with the AVISPA tool [3]. Our transference result (described in the next section) will ensure that the protocol $\tilde{\Pi}$ (obtained from Π by applying our transformation) is secure for an unbounded number of sessions.

4 Transformation of Protocols

In Section 4.1 we define our transformation before we state our main result in Section 4.2 whose proof is postponed to Section 5. Finally, we discuss the tags which are used in our transformation in Section 4.3.

4.1 Our Transformation

Given an input protocol Π, our transformation will compute a new protocol $\tilde{\Pi}$ which consists of two phases. During the first phase, the protocol participants try to agree on some common, dynamically generated, session identifier τ. For this, each participant sends a freshly generated nonce N_i together with his identity A_i to all other participants. (Note that if broadcast is not practical or if not all identities are known to each participant, the message can be sent to some of the participants who forward the message.) At the end of this preamble, each participant computes a session identifier: $\tau = \langle\langle A_1, N_1\rangle, \ldots, \langle A_k, N_k\rangle\rangle$. Note that an active attacker may interfere with this initialization phase and may intercept and replace some of the nonces. Hence, the protocol participants do not necessarily agree on the same session identifier τ after this preamble. In fact, each participant computes his own session identifier, say τ_j. During the second phase, each participant j executes the original protocol in which the dynamically computed identifier is used for tagging each application of a cryptographic primitive. In this phase, when a participant opens an encryption, he will check that the tag is in accordance with the nonces he received during the initialization phase. In particular he can test the presence of his own nonce.

The transformation, using the informal Alice-Bob notation, is described below and relies on the tagging operation that is formally defined in Definition 7.

$$\Pi = \begin{cases} A_{i_1} \to A_{j_1} : m_1 \\ \vdots \\ A_{i_\ell} \to A_{j_\ell} : m_\ell \end{cases} \qquad \tilde{\Pi} = \begin{cases} \text{Phase 1} & \text{Phase 2} \\ A_1 \to All : \langle A_1, N_1\rangle & A_{i_1} \to A_{j_1} : [m_1]_\tau \\ \vdots & \vdots \\ A_k \to All : \langle A_k, N_k\rangle & A_{i_\ell} \to A_{j_\ell} : [m_\ell]_\tau \\ \text{where } \tau = \langle \mathsf{tag}_1, \ldots, \mathsf{tag}_k\rangle \text{ with } \mathsf{tag}_i = \langle A_i, N_i\rangle \end{cases}$$

Note that, the Alice-Bob notation only represents what happens in a normal execution, i.e. with no intervention of the attacker. Of course, in such a situation, the participants agree on the same session identifier τ used in the second phase.

Definition 7 (k-tag, k-tagging). *A k-*tag *is a term $\langle\langle a_1, v_1\rangle, \ldots, \langle a_k, v_k\rangle\rangle$ where each $a_i \in \mathcal{A}$ and each v_i is a term. Let u be a term and* tag *be a k-tag. The k-tagging of u with* tag*, denoted $[u]_{\mathsf{tag}}$, is inductively defined as follows:*

$$\begin{array}{lcll} [\langle u_1, u_2\rangle]_{\mathsf{tag}} & = & \langle [u_1]_{\mathsf{tag}}, [u_2]_{\mathsf{tag}}\rangle & \\ [f(u_1, u_2)]_{\mathsf{tag}} & = & f(\langle \mathsf{tag}, [u_1]_{\mathsf{tag}}\rangle, [u_2]_{\mathsf{tag}}) & \text{for } f \in \{\mathsf{enc}, \mathsf{enca}, \mathsf{sign}\} \\ [\mathsf{h}(u_1)]_{\mathsf{tag}} & = & \mathsf{h}(\langle \mathsf{tag}, [u_1]_{\mathsf{tag}}\rangle) & \\ [u]_{\mathsf{tag}} & = & u & \text{otherwise} \end{array}$$

This notion is extended to sequences of events as expected. We are now able to formally define our transformation.

Definition 8 (Protocol transformation). *Let Π be a k-party protocol such that*

$$\Pi(j) = \lambda x_1^j \ldots \lambda x_k^j . \nu y_1^j \ldots \nu y_{p_j}^j . \mathsf{seq}^j \quad \text{for } 1 \leq j \leq k.$$

and the variables z_i^j ($1 \leq i, j \leq k$) do not appear in Π (which can always be ensured by renaming variables in Π). The transformed protocol $\tilde{\Pi}$ is a k-party protocol defined as follows:

$$\tilde{\Pi}(j) = \lambda x_1^j \ldots \lambda x_k^j . \nu y_1^j \ldots \nu y_{p_j}^j . \nu z_j^j . \tilde{\Pi}^{\mathsf{init}}(j); [\mathsf{seq}^j]_{\tau_j} \quad \text{for } 1 \leq j \leq k$$

where $\tau_j = \langle u_1^j, \ldots, u_k^j\rangle$ with $u_i^j = \langle x_i^j, z_i^j\rangle$, and

$$\tilde{\Pi}^{\mathsf{init}}(j) = \mathsf{rcv}(u_1^j); \ldots; \mathsf{rcv}(u_{j-1}^j); \mathsf{snd}(u_j^j); \mathsf{rcv}(u_{j+1}^j); \ldots; \mathsf{rcv}(u_k^j)$$

In the above definition, the protocol $\tilde{\Pi}^{\mathsf{init}}$ models the initialization phase and the variables z_i^j correspond to the nonces that are exchanged during this phase. In particular for the role j, the variable z_j^j is a freshly generated nonce while the other variables z_i^j, $i \neq j$, are expected to be bound to the other participant's nonces in the receive events. Remember also that the variables x_i^j are the role parameters which correspond to the agents. The tag computed by the j^{th} role in our transformation consists in the concatenation of the $k-1$ nonces received during the initialization phase together with the fresh nonce generated by the role j itself, i.e. z_j^j. We illustrate this transformation on the Needham-Schroeder protocol introduced in Section 2.

Example 5. Consider the Needham-Schroeder protocol described in Example 3. Applying our transformation we obtain a 2-party protocol $\tilde{\Pi}$. The role $\tilde{\Pi}(2)$ is described below. The role $\tilde{\Pi}(1)$ can be obtained in a similar way.

$$\begin{array}{l} \tilde{\Pi}(2) = \lambda x_A \lambda x_B . \nu y' . \nu z_B . \mathsf{rcv}(\langle x_A, z_A\rangle); \mathsf{snd}(\langle x_B, z_B\rangle); \\ \qquad \mathsf{rcv}(\mathsf{enca}(\langle \tau, \langle z', x_A\rangle\rangle, x_B)); \\ \qquad \mathsf{snd}(\mathsf{enca}(\langle \tau, \langle z', y'\rangle\rangle, x_A)); \\ \qquad \mathsf{rcv}(\mathsf{enca}(\langle \tau, y'\rangle, x_B)) \end{array}$$

where $\tau = \langle\langle x_A, z_A\rangle, \langle x_B, z_B\rangle\rangle$. Note that Lowe's famous man-in-the-middle attack [20] described in Example 4 does not exist anymore on $\tilde{\Pi}$.

4.2 Main Theorem

We are now able to state our main transference result.

Theorem 1. *Let Π be a k-party protocol, $\tilde{\Pi}$ be its corresponding transformed protocol, and T_0 be a finite set of ground atoms. Let $m \in St(\Pi(j))$ for some $1 \leq j \leq k$ and $\tilde{m}$ be its counterpart in $\tilde{\Pi}(j)$. Let $\mathsf{CK} = lgKeys(\Pi) \setminus (T_0 \cup \mathcal{K}_\epsilon)$ and assume that $\mathsf{CK} \cap plaintext(\Pi) = \emptyset$, i.e. critical keys do not appear in plaintext.*

If Π preserves the secrecy of m w.r.t. T_0 when considering one honest session of each role, then $\tilde{\Pi}$ preserves the secrecy of $\tilde{m}$ w.r.t. T_0.

Our result states that if the compiled protocol admits an attack that may involve several honest and dishonest sessions, then there exists an attack which only requires one honest session of each role (and no dishonest sessions). The situation is however slightly more complicated than it may seem at first sight since there is an infinite number of honest sessions, which one would need to verify separately. Actually we can avoid this combinatorial explosion thanks to the following well-known result [9]: when verifying secrecy properties it is sufficient to consider one single honest agent (which is allowed to "talk to herself"). Hence we can instantiate all the parameters with the same agent $a \in \mathcal{A} \setminus \{\epsilon\}$.

Our dynamic tagging is useful to avoid interaction between different sessions of the same role in a protocol execution and allows us for instance to prevent man-in-the-middle attacks (see Example 5). A more detailed discussion showing that *static tags* are not sufficient follows in Section 4.3. As stated in Theorem 1 we need also to forbid long-term secrets in plaintext position (even under an encryption). Note that this condition is generally satisfied by protocols and considered as a prudent engineering practice.

4.3 Other Ways of Tagging

We have also considered an alternative, slightly different transformation that does not include the identities in the tag, i.e., the tag is simply the sequence of nonces. In that case we obtain a different result: if a protocol admits an attack then there exists an attack which only requires one (not necessarily honest) session for each role. In this case, we need to additionally check for attacks that involve a session engaged with the attacker. On the example of the Needham-Schroeder protocol the man-in-the-middle attack is not prevented by this weaker tagging scheme. However, the result requires one to also consider one dishonest session for each role, hence including the attack scenario. In both cases, it is important for the tags to be *collaborative*, *i.e.* all participants do contribute by adding a fresh nonce.

Finally, different kinds of tags have also been considered in [2,6,23]. However these tags are *static* and have a different aim. While our dynamic tagging scheme avoids confusing messages from different sessions, these static tags avoid confusing different messages inside a same session and do not prevent that a same message is reused in two different sessions. Under some additional assumptions (e.g. no temporary secret, no ciphertext forwarding), several decidability results [24,21] have been obtained by showing that it is sufficient to consider one session per role. But those results can not

deal with protocols such as the Yahalom protocol or protocols which rely on a temporary secret. In the framework we consider here, the question whether such static tags would be sufficient to obtain decidability is still an open question (see [2]). In a similar way, static tags have also been used by Heather et al. [18] to avoid type confusion attacks.

5 Proof of Our Main Result

The proof of our main result is closely tied to a particular procedure for solving constraint systems (see [8,13,10]). We therefore first give a brief description of this procedure before outlining the proof itself.

5.1 Constraint Solving Procedure

The procedure we consider for constraint solving uses the simplification rules of Figure 2 to transform a given constraint system into another, simpler one. Such a simplification step is denoted $C \rightsquigarrow_\sigma C'$ where σ is a substitution that has been applied to C during this step (when omitted it implicitly refers to the identity function). We also write $C \rightsquigarrow_\sigma^n C'$ for a sequence of n steps where σ is the composition of the substitutions applied at each step.

Our constraint solving procedure is very similar to the ones presented in [8,13,10,14]: soundness, completeness and termination can be proved in a similar way. This means that the procedure always terminates after a finite number of steps resulting either in $\bot$ when no solution exists or in a constraint system in *solved form*. A constraint system is in solved form when the right-hand side of each constraint is a variable. In that case the constraint system can be trivially satisfied. Moreover, we assume that rules $\mathsf{R}_2, \mathsf{R}_3, \mathsf{R}_4, \mathsf{R}_5$, and R_6 are applied in priority, i.e. before any instance of the rule R_1. It is easy to see that the procedure remains complete.

For the purpose of our proof, we decorate each term t by a pair (r, s) which denotes the role number and the session identifier in which t originated. The resulting term $t^{(r,s)}$ is called a labeled term. By convention, terms in T_0 (the initial attacker knowledge) are labeled with $(0, 0)$. These decorations do not influence the procedure but provide additional information that is useful in the proof. We could have added these decorations before, but it would increase notational clutter and harm readability.
We extend all notations defined on terms to labeled terms, by providing a session identifier as an additional argument: e.g. $vars(T, sid) = \bigcup_{t^{(r,sid)} \in T} vars(t)$.

5.2 Proof of Theorem 1

Theorem 1 is proved by contradiction. Assume that $\tilde{\Pi}$ admits an attack. This means that there exists a scenario sc, a function $\alpha : \mathbb{N} \to \mathcal{A}^k$ and an honest session sid_h such that the associated constraint system C (according to Definition 6) is satisfiable. We will prove that the constraint system C' associated to sc' (the subsequence of sc where we only consider some particular sessions, say S, chosen according to the tag τ_{sid_h}

$$\mathsf{R}_1 : C \wedge T \Vdash u^{(r,sid)} \rightsquigarrow C \qquad \text{if } T \cup \{x \mid T' \Vdash x \in C, T' \subsetneq T\} \vdash u$$

$$\mathsf{R}_2 : C \wedge T \Vdash u^{(r,sid)} \rightsquigarrow_\sigma C\sigma \wedge T\sigma \Vdash u\sigma^{(r,sid)}$$
$$\text{if } \sigma = \mathrm{mgu}(t,u) \text{ where } t \in St(T), t \neq u, t, u \text{ are neither variables nor pairs}$$

$$\mathsf{R}_3 : C \wedge T \Vdash u^{(r,sid)} \rightsquigarrow_\sigma C\sigma \wedge T\sigma \Vdash u\sigma^{(r,sid)}$$
$$\text{if } \sigma = \mathrm{mgu}(t_1,t_2), t_1, t_2 \in St(T), t_1 \neq t_2, t_1, t_2 \text{ are neither variables nor pairs}$$

$$\mathsf{R}_4 : C \wedge T \Vdash u^{(r,sid)} \rightsquigarrow_\sigma C\sigma \wedge T\sigma \Vdash u\sigma^{(r,sid)}$$
$$\text{if } \sigma = \mathrm{mgu}(t_2,t_3), \mathsf{enca}(t_1,t_2) \in St(T), \mathsf{priv}(t_3) \in (plaintext(T) \cup \{\mathsf{priv}(\epsilon)\}), \ t_2 \neq t_3$$

$$\mathsf{R}_5 : C \wedge T \Vdash u^{(r,sid)} \rightsquigarrow \bot \qquad \text{if } vars(T \cup \{u\}) = \emptyset \text{ and } T \not\vdash u$$

$$\mathsf{R}_6 : C \wedge T \Vdash f(u_1,\ldots,u_n)^{(r,sid)} \rightsquigarrow C \wedge \{T \Vdash u_i^{(r,sid)} \mid 1 \leq i \leq n\}$$
$$\text{for } f \in \{\langle\rangle, \mathsf{enc}, \mathsf{enca}, \mathsf{sign}, \mathsf{h}\}$$

Fig. 2. Simplification rules

involved in the cryptographic subterms of the honest session sid_h) and α, is also satisfiable. Intuitively $sid \in S$ if and only if the nonce generated during the initialization phase of the session sid appears in tag τ_{sid_h} at the expected position, i.e. at the r^{th} position where r is the role associated to the session identifier sid. Initially, we have that $C' = C|_S$ according to the following definition.

Definition 9 (Constraint system $C|_S$). *Let C be a constraint system and S a set of session identifiers, we define the restriction of C to S as follows*

$$C|_S := \{T|_S \Vdash u^{(r,s)} \mid s \in S \text{ and } (T \Vdash u^{(r,s)}) \in C\},$$

where $T|_S = \{v^{(r,s)} \in T \mid s \in S \cup \{0\}\}$.

We want to ensure that the simplification steps are stable by restriction to some well-chosen set S of sessions (see Lemma 2). This property does not hold for general constraint systems but only for *well-formed* constraint systems. This notion relies on some additional definitions.

Definition 10 (k-tagged). *A term t is* k-tagged *if all its cryptographic subterms are tagged with a k-tag , i.e. $\forall u \in CryptSt(t)$, $\exists \mathsf{tag}, u_1, \ldots, u_n.\ u = f(\langle \mathsf{tag}, u_1\rangle, \ldots, u_n)$ where $f \in \{\mathsf{enc}, \mathsf{enca}, \mathsf{sign}, \mathsf{h}\}$.*

We denote by $tags(t)$ the set of k-tags which occur in a tagging position in t. Given a set T of k-tagged labeled terms, $tags(T, sid) = \bigcup_{t^{(i,sid)} \in T} tags(t)$.

Definition 11 (Well-formed). *A constraint system $C = (T_i \Vdash u_i)_{1 \leq i \leq n}$ is* well-formed *w.r.t. a set T of k-tagged labeled terms if the following hold:*

1. $\mathsf{maxlhs}(C) \subseteq T$ *and* $\mathsf{rhs}(C) \subseteq St(T)$*;*
2. *the constraint system C satisfies the* plaintext origination property*, i.e. if $x \in vars(plaintext(T_i))$ then $\exists\, j < i$ such that $x \in vars(plaintext(u_j))$;*
3. *for all sid we have that $|tags(T, sid)| \leq 1$;*
4. *for all sid_1, sid_2 such that $tags(T, sid_1) \neq tags(T, sid_2)$, we have that $vars(T, sid_1) \cap vars(T, sid_2) = \emptyset \ \wedge\ fresh(T, sid_1) \cap fresh(T\, sid_2) = \emptyset$.*

Intuitively, Condition 1 states that the terms in C are k-tagged. Condition 2 ensures that any variable appearing as a plaintext has been previously received in a plaintext position (this is ensured thanks to the condition 2 of Definition 2). Condition 3 says that all terms that originated in the same session have the same tag. Finally, Condition 4 ensures that sessions that are currently tagged in different ways in C use different variables and different nonces. Note that terms issued from different sessions are not necessarily tagged differently. First, we show that the simplification rules maintain well-formedness.

Lemma 1. *Let T be a set of k-tagged labeled terms, and C be a constraint system well-formed w.r.t. T. Let D be a constraint system, σ be a substitution and n be an integer such that $C \rightsquigarrow^n_\sigma D$. Then, we have that D is well-formed w.r.t. $T\sigma$ and, for any session sid, we have that $tags(T\sigma, sid) = (tags(T, sid))\sigma$.*

Relying on Lemma 1 we show that there exists a derivation from $C|_S$ to a constraint system in solved form, i.e. the existence of an attack involving only sessions in S.

Lemma 2. *Let CK_0 be a set of ground atoms such that $\mathsf{CK}_0 \cap \mathcal{K}_\epsilon = \emptyset$, T be a set of k-tagged labeled terms and C be a constraint system well-formed w.r.t. T and such that*

1. *$(lgKeys(C) \setminus \mathsf{CK}_0) \subseteq \mathsf{minlhs}(C)$ and those terms are labeled with $(0, 0)$, and*
2. *$\mathsf{CK}_0 \cap plaintext(\mathsf{maxlhs}(C)) = \emptyset$.*

Let D be a satisfiable constraint system, σ be a substitution and n be an integer such that $C \rightsquigarrow^n_\sigma D$. Let tag be a k-tag, $Sid(\mathsf{tag}) = \{sid \mid tags(T\sigma, sid) = \mathsf{tag}\}$. If $(\mathsf{maxlhs}(C) \Vdash u^{(r,sid)}) \in C$ for some u, r and $sid \in Sid(\mathsf{tag})$ then there exists $m \leq n$ such that $C|_{Sid(\mathsf{tag})} \rightsquigarrow^m_{\sigma|_Y} D|_{Sid(\mathsf{tag})}$, where $Y = \bigcup_{sid \in Sid(\mathsf{tag})} vars(T, sid)$.

This lemma is proved by induction. The proof is technical and the details can be found in [1]. We consider the different rules and distinguish several cases depending on whether the terms involved are labeled with a session identifier in S or not. For instance, the rules R_5 (resp. R_6) are mimicked by using the same instance of the same rule when the labeled term $u^{(r,sid)}$ (right-hand side of the constraint) is such that $sid \in S$. Otherwise, we keep the constraint system unchanged. For the rule R_2 (resp. R_3) the key point is that terms which are tagged differently cannot be unified and do not share any variables nor fresh names (this is due to well-formedness). Thus, the unifier σ used in this step involved two terms labeled by sid_1 and sid_2 that are either both in S or both not in S. This is due to the fact that, after application of σ, these two terms will be tagged in the same way and thus by definition of S, have the same status. If both are in S, we can apply the same rule. If none of them is in S, we show that σ has no effect and we keep the constraint system unchanged. The case of the rules R_1 and R_4 can also be proved in a similar way.

In order to pursue the proof of Theorem 1, we apply Lemma 2 on the derivation $C \rightsquigarrow^n_\sigma D$ witnessing the existence of an attack on $\tilde{\Pi}$ and we consider $S = \{sid \mid tags(T\sigma, sid) = tags(T\sigma, sid_h)\}$, i.e. the sessions that are tagged in the same way that sid_h. We obtain that $C|_S$ can also reach a constraint system in solved form, namely $D|_S$. Moreover, the satisfiability of $C|_S$ witnesses the fact that there is an attack on $\tilde{\Pi}$ that only involves sessions in S. In order to conclude, it remains to show that:

1. S *does not contain two distinct sessions that execute the same role*. Intuitively, this comes from the fact that sessions in S are tagged in the same way (after application of σ) and this is not possible for two distinct sessions that execute the same role. Indeed, the fresh nonce generated by different sessions of the same role ensures that their tags are distinct.
2. S *only contains honest sessions*. First sid_h is an honest session by definition of the secrecy property. Second, since the names of the agents engaged in a role occur in the tag and sessions in S are tagged in the same way as the session sid_h, we conclude that this property is also true for any $sid \in S$.

Thus, there is an attack on $\tilde{\Pi}$ that involves at most one honest session of each role. To conclude, it is easy to see that this attack can also be mounted on the protocol Π.

6 Future Work

Our current result applies to transfer secrecy properties. As future work we foresee to extend the scope of our result to other security properties, e.g. authentication or more challenging equivalence based properties. We also plan to extend the result to other intruder theories. We foresee that such results require new proof methods which are not based on the decision procedure as in this paper, but directly on the semantics. Another challenging topic for future research is to obtain more fine-grained characterizations of decidable classes of protocols for an unbounded number of sessions. The new insights gained by our work seem to be a good starting point to extract the conditions needed to reduce the security for an unbounded number of sessions to a finite number of sessions.

Acknowledgments. We would like to thank Yassine Lakhnech for discussions that initiated this work as well as Hubert Comon-Lundh, Véronique Cortier, Joshua Guttman and Ralf Küsters for their helpful comments.

References

1. Arapinis, M., Delaune, S., Kremer, S.: From one session to many: Dynamic tags for security protocols. Research Report LSV-08-20, ENS Cachan, France, 30 pages (June 2008)
2. Arapinis, M., Duflot, M.: Bounding messages for free in security protocols. In: Arvind, V., Prasad, S. (eds.) FSTTCS 2007. LNCS, vol. 4855, pp. 376–387. Springer, Heidelberg (2007)
3. Armando, A., et al.: The Avispa tool for the automated validation of internet security protocols and applications. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 281–285. Springer, Heidelberg (2005)
4. Beauquier, D., Gauche, F.: How to guarantee secrecy for cryptographic protocols. CoRR, abs/cs/0703140 (2007)
5. Bellare, M., Canetti, R., Krawczyk, H.: A modular approach to the design and analysis of authentication and key exchange protocols (extended abstract). In: Proc. 30th Annual ACM Symposium on the Theory of Computing (STOC 1998), pp. 419–428. ACM Press, New York (1998)
6. Blanchet, B., Podelski, A.: Verification of cryptographic protocols: Tagging enforces termination. In: Gordon, A.D. (ed.) FOSSACS 2003. LNCS, vol. 2620, pp. 136–152. Springer, Heidelberg (2003)

7. Clark, J., Jacob, J.: A survey of authentication protocol literature (1997)
8. Comon, H.: Résolution de contraintes et recherche d'attaques pour un nombre borné de sessions, http://www.lsv.ens-cachan.fr/~comon/CRYPTO/bounded.ps
9. Comon-Lundh, H., Cortier, V.: Security properties: two agents are sufficient. Science of Computer Programming 50(1-3), 51–71 (2004)
10. Cortier, V., Delaitre, J., Delaune, S.: Safely composing security protocols. In: Arvind, V., Prasad, S. (eds.) FSTTCS 2007. LNCS, vol. 4855, pp. 352–363. Springer, Heidelberg (2007)
11. Cortier, V., Delaune, S.: Safely composing security protocols. Research Report LSV-08-06, ENS Cachan, France, 39 pages (March 2008)
12. Cortier, V., Warinschi, B., Zălinescu, E.: Synthesizing secure protocols. In: Biskup, J., López, J. (eds.) ESORICS 2007. LNCS, vol. 4734, pp. 406–421. Springer, Heidelberg (2007)
13. Cortier, V., Zălinescu, E.: Deciding key cycles for security protocols. In: Hermann, M., Voronkov, A. (eds.) LPAR 2006. LNCS, vol. 4246, pp. 317–331. Springer, Heidelberg (2006)
14. Delaune, S.: Note: Constraint solving procedure, http://www.lsv.ens-cachan.fr/Publis/PAPERS/PDF/CDD-fsttcs07-addendum.pdf
15. Dolev, D., Even, S., Karp, R.M.: On the security of ping-pong protocols. In: Proc. Advances in Cryptology (CRYPTO 1982), pp. 177–186 (1982)
16. Dolev, D., Yao, A.C.: On the security of public key protocols. In: Proc. of the 22nd Symposium on Foundations of Computer Science (FOCS 1981). IEEE Comp. Soc. Press, Los Alamitos (1981)
17. Durgin, N., Lincoln, P., Mitchell, J., Scedrov, A.: Undecidability of bounded security protocols. In: Proc. Workshop on Formal Methods and Security Protocols (1999)
18. Heather, J., Lowe, G., Schneider, S.: How to prevent type flaw attacks on security protocols. In: Proc. 13th Computer Security Foundations Workshop (CSFW 2001), pp. 255–268. IEEE Comp. Soc. Press, Los Alamitos (2000)
19. Katz, J., Yung, M.: Scalable protocols for authenticated group key exchange. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 110–125. Springer, Heidelberg (2003)
20. Lowe, G.: Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In: Margaria, T., Steffen, B. (eds.) TACAS 1996. LNCS, vol. 1055, pp. 147–166. Springer, Heidelberg (1996)
21. Lowe, G.: Towards a completeness result for model checking of security protocols. Journal of Computer Security 7(1) (1999)
22. Millen, J., Shmatikov, V.: Constraint solving for bounded-process cryptographic protocol analysis. In: Proc. 8th ACM Conference on Computer and Communications Security (CCS 2001). ACM Press, New York (2001)
23. Ramanujam, R., Suresh, S.P.: Tagging makes secrecy decidable for unbounded nonces as well. In: Pandya, P.K., Radhakrishnan, J. (eds.) FSTTCS 2003. LNCS, vol. 2914, pp. 363–374. Springer, Heidelberg (2003)
24. Ramanujam, R., Suresh, S.P.: Decidability of context-explicit security protocols. Journal of Computer Security 13(1), 135–165 (2005)
25. Rusinowitch, M., Turuani, M.: Protocol insecurity with finite number of sessions and composed keys is NP-complete. Theoretical Computer Science 299(1-3), 451–475 (2003)

A Conditional Logical Framework*

Furio Honsell, Marina Lenisa, Luigi Liquori, and Ivan Scagnetto

INRIA, France & UNIUD, Italy
{honsell,lenisa,scagnett}@dimi.uniud.it,Luigi.Liquori@inria.fr

Abstract. The *Conditional Logical Framework* LF_κ is a variant of the Harper-Honsell-Plotkin's Edinburgh Logical Framemork LF. It features a generalized form of λ-abstraction where β-reductions fire *under the condition* that the argument satisfies a *logical predicate*. The key idea is that the type system *memorizes* under what conditions and where reductions have yet to fire. Different notions of β-reductions corresponding to different predicates can be combined in LF_κ. The framework LF_κ subsumes, by simple instantiation, LF (in fact, it is also a subsystem of $\mathsf{LF}!$), as well as a large class of new generalized conditional λ-calculi. These are appropriate to deal smoothly with the side-conditions of both Hilbert and Natural Deduction presentations of Modal Logics. We investigate and characterize the metatheoretical properties of the calculus underpinning LF_κ, such as subject reduction, confluence, strong normalization.

1 Introduction

The Edinburgh Logical Framework LF of [HHP93] was introduced both as a general theory of logics and as a metalanguage for a generic proof development environment. In this paper, we consider a variant of LF, called *Conditional Logical Framework* LF_κ, which allows to deal uniformly with logics featuring *side-conditions* on the application of inference rules, such as *Modal Logics*. We study the language theory of LF_κ and we provide proofs for subject reduction, confluence, and strong normalization. By way of example, we illustrate how special instances of LF_κ allow for smooth encodings of Modal Logics both in Hilbert and Natural Deduction style.

The motivation for introducing LF_κ is that the type system of LF is too coarse as to the "side conditions" that it can enforce on the application of rules. Rules being encoded as functions from proofs to proofs and rule application simply encoded as lambda application, there are only roundabout ways to encode provisos, even as simple as that appearing in a *rule of proof*. Recall that a rule of proof can be applied only to premises which do not depend on any assumption, as opposed to a *rule of derivation* which can be applied everywhere. Also rules which appear in many natural deduction presentations of Modal and Program Logics are very problematic in standard LF. Many such systems feature rules which can be applied only to premises which depend solely on assumptions of a particular shape [CH84], or whose derivation has been carried out using only certain sequences of rules. In general, Modal, Program, Linear or Relevance

* Supported by AEOLUS FP6-IST-FET Proactive.

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 143–157, 2008.

Logics appear to be encodable in LF only encoding a very heavy machinery, which completely rules out any natural Curry-Howard paradigm, see *e.g.* [AHMP98]. As we will see for Modal Logics, LF_κ allows for much simpler encodings of such rules, which open up promising generalizations of the proposition-as-types paradigm.

The idea underlying the Conditional Logical Framework LF_κ is inspired by the Honsell-Lenisa-Liquori's *General Logical Framework* GLF see [HLL07], where we proposed a uniform methodology for extending LF, which allows to deal with pattern matching and restricted λ-calculi. The key idea, there, is to separate two different notions that are conflated in the original LF. As already mentioned, much of the rigidity of LF arised from the fact that β-reduction can be applied always in full generality. One would like to fire a β-reduction under certain conditions on typed terms, but the type system is not rich enough to be able to express such restrictions smoothly. What we proposed in [HLL07] is to use as type of an application, in the term application rule, $(\mathsf{O{\cdot}Appl})$ below, not the type which is obtained by carrying out directly in the metalanguage the substitution of the argument in the type, but a new form of type which simply records the information that such a reduction can be carried out. An application of the Type Conversion Rule can then recover, if possible, the usual effect of the application rule. This key idea leads to the following object application rule:

$$\frac{\Gamma \vdash M : \Pi x{:}A.B \quad \Gamma \vdash N : A}{\Gamma \vdash M\,N : (\lambda x{:}A.B)\,N}$$

Once this move has been made, we have a means of annotating in a type the information that a reduction *is waiting to be carried out in the term*. If we take seriously this move, such a type need not be necessarily definitionally equal to the reduced one as in the case of LF. Without much hassle we have a principled and natural way of typing calculi featuring generalized or restricted forms of β-reduction, which *wait for some condition to be satisfied before they can fire*. Furthermore, such calculi can be used for underpinning new powerful Logical Frameworks, where all the extra complexity in terms can be naturally tamed using the expressive power of the new typing system. Once this program is carried out in a sufficiently modular form, we have a full-fledged Logical Framework.

More specifically, in LF_κ we consider a new form of λ and corresponding Π abstraction, *i.e.* $\lambda_{\mathcal{P}} x{:}A.M$ and $\Pi_{\mathcal{P}} x{:}A.M$, where $\mathcal{P}$ is a predicate, which ranges over a suitable set of predicates. The reduction $(\lambda_{\mathcal{P}} x{:}A.M)\,N$ fires only if the predicate $\mathcal{P}$ holds on N, and in this case the redex progresses, as usual, to $M[N/x]$. Therefore the final object application rule in LF_κ will be:

$$\frac{\Gamma \vdash_\Sigma M : \Pi_{\mathcal{P}} x{:}A.B \quad \Gamma \vdash_\Sigma N : A}{\Gamma \vdash_\Sigma M\,N : (\lambda_{\mathcal{P}} x{:}A.B)\,N}$$

In this rule a type where a reduction is "stuck", if the predicate $\mathcal{P}$ is not true on N, is assigned to an object application. However, when we view this object as a subterm of another term, such reduction could become allowed in the future, after other reductions are performed in the term, which provide substitutions for N. In LF_κ more predicates can be combined. LF_κ subsumes standard LF, which is recovered by considering the trivial predicate that is constantly true.

Historically, the idea of introducing stuck-reduction in objects and types, in the setting of higher-order term rewriting systems with sophisticated pattern-matching capabilities, was first introduced in Cirstea-Kirchner-Liquori's Rho-cube [CKL01b], in order to design a hierarchy of type systems for the untyped Rewriting Calculus of [CKL01a], and then it was generalized to a more general framework of Pure Type

Systems with Patterns [BCKL03]. This typing protocol was essential to preserve the strong normalization of typable terms, as proved in [HLL07]. The idea underlying the Conditional Logical Framework LF_K is the same exploited in [HLL07] for the General Logical Framework GLF. However, there is an important difference between the two frameworks in the definition of predicates. On one hand, predicates in [HLL07] are used both to determine whether β-reduction fires and to compute a substitution, while in the present paper they are used only to determine whether β-reduction fires. On the other hand, in [HLL07] predicates are defined on terms, while here they are defined on typed judgments. This adds extra complexity both in the definition of the system and in the study of its properties, but it greatly simplifies the treatment of Modal Logics and of other situations where conditions depending on types have to be expressed.

Apart from Modal Logics, we believe that our Conditional Logical Framework could also be very helpful in modeling dynamic and reactive systems: for example bio-inspired systems, where reactions of chemical processes take place only provided some extra structural or temporal conditions; or process algebras, where often no assumptions can be made about messages exchanged through the communication channels. Indeed, it could be the case that a redex, depending on the result of a communication, can remain stuck until a "good" message arrives from a given channel, firing in that case an appropriate reduction (this is a common situation in many protocols, where "bad" requests are ignored and "good ones" are served). Such dynamical (run-time) behaviour could be hardly captured by a rigid type discipline, where bad terms and hypotheses are ruled out *a priori*, see *e.g.* [NPP08].

In this paper we develop all the metatheory of LF_K. In particular, we prove subject reduction, strong normalization, confluence; this latter under the sole assumption that the various predicate reductions nicely combine, *i.e.* no reduction can prevent a redex, which could fire, from firing after the reduction. Since β-reduction in LF_K is defined only on typed terms, in order to prove subject reduction and confluence, we need to devise a new approach, alternative to the one in [HHP93]. Our approach is quite general, and in particular it yields alternative proofs for the original LF.

In conclusion, the work on LF_K carried out in this paper is valuable in three ways. First, being LF_K so general, the results in this paper potentially apply to a wide range of Logical Frameworks, therefore many fundamental results are proved only once and uniformly for all systems. Secondly, the LF_K approach is useful in view of implementing a "telescope" of systems, since it provides relatively simple sufficient conditions to test whether a potential extension of the framework is safe. Thirdly, LF_K can suggest appropriate extensions of the proposition-as-types paradigm to a wider class of logics.

Synopsis. In Section 2, we present the syntax of LF_K, its type system, and the predicate reduction. In Section 3, we present instantiations of LF_K to known as well as to new calculi, and we show how to encode smoothly Modal Logics. The LF_K's metatheory is carried out in Section 4. Conclusions and directions for future work appear in Section 5. Proofs appear in a Web Appendix available at the author's web pages.

2 The System

Syntax. In the following definition, we introduce the LF_{K} pseudo-syntax for kinds, families, objects, signatures and contexts.

Definition 1 (LF_{K} **Pseudo-syntax)**

$\Sigma \in \mathcal{S}$	$\Sigma ::= \emptyset \mid \Sigma, a{:}K \mid \Sigma, f{:}A$	*Signatures*
$\Gamma, \Delta \in \mathcal{C}$	$\Gamma ::= \emptyset \mid \Gamma, x{:}A$	*Contexts*
$K \in \mathcal{K}$	$K ::= \mathsf{Type} \mid \Pi_{\mathcal{P}} x{:}A.K \mid \lambda_{\mathcal{P}} x{:}A.K \mid K\ M$	*Kinds*
$A, B, C \in \mathcal{F}$	$A ::= a \mid \Pi_{\mathcal{P}} x{:}A.B \mid \lambda_{\mathcal{P}} x{:}A.B \mid A\ M$	*Families*
$M, N, Q \in \mathcal{O}$	$M ::= f \mid x \mid \lambda_{\mathcal{P}} x{:}A.M \mid M\ N$	*Objects*

where a, f are typed constants standing for fixed families and terms, respectively, and $\mathcal{P}$ is a predicate ranging over a set of predicates, which will be specified below.

LF_{K} is parametric over a set of predicates of a suitable shape. Such predicates are defined on typing judgments, and will be discussed in the section introducing the type system.

Notational conventions and auxiliary definitions. Let "T" range over any term in the calculus (kind, family, object). The abstractions $\checkmark_{\mathcal{P}} x{:}A.T$ ($\checkmark \in \{\lambda, \Pi\}$) bind the variable x in T. Domain $\mathsf{Dom}(\Gamma)$ and codomain $\mathsf{CoDom}(\Gamma)$ are defined as usual. Free $\mathsf{Fv}(T)$ and bound $\mathsf{Bv}(T)$ variables are defined as usual. As usual, we suppose that, in the context $\Gamma, x{:}T$, the variable x does not occur free in Γ and T. We work modulo α-conversion and Barendregt's hygiene condition.

Type System. LF_{K} involves type judgments of the following shape:

$\Sigma\ \mathsf{sig}$	Σ is a valid signature
$\vdash_{\Sigma} \Gamma$	Γ is a valid context in Σ
$\Gamma \vdash_{\Sigma} K$	K is a kind in Γ and Σ
$\Gamma \vdash_{\Sigma} A : K$	A has kind K in Γ and Σ
$\Gamma \vdash_{\Sigma} M : A$	M has type A in Γ and Σ
$\Gamma \vdash_{\Sigma} T \mapsto_{\beta} T' (: T'')$	T reduces to T' in Γ, Σ (and T'')
$\Gamma \vdash_{\Sigma} T =_{\beta} T' (: T'')$	T converts to T' in Γ, Σ (and T'')

The typing rules of LF_{K} are presented in Figure 1. As remarked in the introduction, rules ($\mathsf{F{\cdot}Appl}$) and ($\mathsf{O{\cdot}Appl}$) do not utilize metasubstitution as in standard LF, but rather introduce an explicit type redex. Rules ($\mathsf{F{\cdot}Conv}$) and ($\mathsf{O{\cdot}Conv}$) allow to recover the usual rules, if the reduction fires.

Typed Operational Semantics. The "type driven" operational semantics is presented in Figure 2, where the most important rule is ($\mathsf{O{\cdot}Red}$), the remaining ones being the contextual closure of β-reduction. For lack of space we omit similar rules for kinds and constructors. According to rule ($\mathsf{O{\cdot}Red}$), reduction is allowed only if the argument in

Signature and Context rules

$$\frac{}{\emptyset \text{ sig}}\ (\mathsf{S{\cdot}Empty}) \qquad \frac{\Sigma \text{ sig}}{\vdash_\Sigma \emptyset}\ (\mathsf{C{\cdot}Empty})$$

$$\frac{\Sigma \text{ sig} \quad \vdash_\Sigma K \quad a \notin \mathsf{Dom}(\Sigma)}{\Sigma, a{:}K \text{ sig}}\ (\mathsf{S{\cdot}Kind})$$

$$\frac{\Sigma \text{ sig} \quad \vdash_\Sigma A : \mathsf{Type} \quad f \notin \mathsf{Dom}(\Sigma)}{\Sigma, f{:}A \text{ sig}}\ (\mathsf{S{\cdot}Type})$$

$$\frac{\vdash_\Sigma \Gamma \quad \Gamma \vdash_\Sigma A : \mathsf{Type} \quad x \notin \mathsf{Dom}(\Gamma)}{\vdash_\Sigma \Gamma, x{:}A}\ (\mathsf{C{\cdot}Type})$$

$$\frac{\Gamma, x{:}A \vdash_\Sigma B : \mathsf{Type}}{\Gamma \vdash_\Sigma \Pi_{\mathcal{P}} x{:}A.B : \mathsf{Type}}\ (\mathsf{F{\cdot}Pi})$$

$$\frac{\Gamma, x{:}A \vdash_\Sigma B : K}{\Gamma \vdash_\Sigma \lambda_{\mathcal{P}} x{:}A.B : \Pi_{\mathcal{P}} x{:}A.K}\ (\mathsf{F{\cdot}Abs})$$

$$\frac{\Gamma \vdash_\Sigma A : \Pi_{\mathcal{P}} x{:}B.K \quad \Gamma \vdash_\Sigma N : B}{\Gamma \vdash_\Sigma A\, N : (\lambda_{\mathcal{P}} x{:}B.K)\, N}\ (\mathsf{F{\cdot}Appl})$$

$$\frac{\Gamma \vdash_\Sigma A : K' \quad \Gamma \vdash_\Sigma K \quad \Gamma \vdash_\Sigma K =_\beta K'}{\Gamma \vdash_\Sigma A : K}\ (\mathsf{F{\cdot}Conv})$$

Kind rules

$$\frac{\vdash_\Sigma \Gamma}{\Gamma \vdash_\Sigma \mathsf{Type}}\ (\mathsf{K{\cdot}Type})$$

$$\frac{\Gamma, x{:}A \vdash_\Sigma K}{\Gamma \vdash_\Sigma \Pi_{\mathcal{P}} x{:}A.K}\ (\mathsf{K{\cdot}Pi})$$

$$\frac{\Gamma, x{:}A \vdash_\Sigma K}{\Gamma \vdash_\Sigma \lambda_{\mathcal{P}} x{:}A.K}\ (\mathsf{K{\cdot}Abs})$$

$$\frac{\Gamma \vdash_\Sigma \lambda_{\mathcal{P}} x{:}A.K \quad \Gamma \vdash_\Sigma N : A}{\Gamma \vdash_\Sigma (\lambda_{\mathcal{P}} x{:}A.K)\, N}\ (\mathsf{K{\cdot}Appl})$$

Family rules

$$\frac{\vdash_\Sigma \Gamma \quad a{:}K \in \Sigma}{\Gamma \vdash_\Sigma a : K}\ (\mathsf{F{\cdot}Const})$$

Object rules

$$\frac{\vdash_\Sigma \Gamma \quad x{:}A \in \Gamma}{\Gamma \vdash_\Sigma x : A}\ (\mathsf{O{\cdot}Var})$$

$$\frac{\vdash_\Sigma \Gamma \quad f{:}A \in \Sigma}{\Gamma \vdash_\Sigma f : A}\ (\mathsf{O{\cdot}Const})$$

$$\frac{\Gamma, x{:}A \vdash_\Sigma M : B}{\Gamma \vdash_\Sigma \lambda_{\mathcal{P}} x{:}A.M : \Pi_{\mathcal{P}} x{:}A.B}\ (\mathsf{O{\cdot}Abs})$$

$$\frac{\Gamma \vdash_\Sigma M : \Pi_{\mathcal{P}} x{:}A.B \quad \Gamma \vdash_\Sigma N : A}{\Gamma \vdash_\Sigma M\, N : (\lambda_{\mathcal{P}} x{:}A.B)\, N}\ (\mathsf{O{\cdot}Appl})$$

$$\frac{\Gamma \vdash_\Sigma M : A \quad \Gamma \vdash_\Sigma B : \mathsf{Type} \quad \Gamma \vdash_\Sigma A =_\beta B : \mathsf{Type}}{\Gamma \vdash_\Sigma M : B}\ (\mathsf{O{\cdot}Conv})$$

Fig. 1. LF_{K} Type System

the context satisfies the predicate $\mathcal{P}$. In this sense, reduction becomes "conditioned" by $\mathcal{P}$. In LF_{K}, we can combine more predicate reductions, *i.e.*, we can define and combine *several predicates* guarding β-reduction, whose shape is as follows. Each predicate is determined by a set $\mathcal{A}$ of families (types), and the intended meaning is that it holds on a typed judgment $\Gamma \vdash_\Sigma M : A$ and a set of variables $\mathcal{X} \subseteq \mathsf{Dom}(\Gamma)$ if "$\Gamma \vdash_\Sigma M : A$ *is derivable and all the free variables in* M *which are in* $\mathcal{X}$ *appear in subterms typable with a type in* $\mathcal{A}$". This intuition is formally stated in the next definition.

Definition 2 (Good families (types) and predicates)
Let $\mathcal{A} \subseteq \mathcal{F}$ be a set of families. This induces a predicate $\mathcal{P}_{\mathcal{A}}$ (denoted by $\mathcal{P}$, for simplicitly), defined on typed judgments $\Gamma \vdash M : A$ and sets $\mathcal{X}$ such that $\mathcal{X} \subseteq \mathsf{Dom}(\Gamma)$. The truth table of $\mathcal{P}$ appears in Figure 3.
We call good *a predicate $\mathcal{P}$ defined as above, and* good types *the set of types in $\mathcal{A}$ inducing it.*

The following lemma states formally the intended meaning of our predicates:

Lemma 1 ($\mathcal{P}$ Satisfiability)
Given a predicate $\mathcal{P} \in \mathcal{L}$ induced by a set of families (types) $\mathcal{A}$, $\mathcal{P}$ holds on a typed judgment $\Gamma \vdash_\Sigma M : B$ and a set of variables $\mathcal{X} \subseteq \mathsf{Dom}(\Gamma)$, if $\Gamma \vdash_\Sigma M : B$ is derivable and all the free variables in M which are in $\mathcal{X}$ appear in subterms typable with a type in $\mathcal{A}$.

$$\frac{\begin{array}{l}\Gamma \vdash_\Sigma (\lambda_{\mathcal{P}} x{:}A.M)\, N : C \\ \Gamma \vdash_\Sigma M[N/x] : C \qquad \mathcal{P}(\mathsf{Fv}(N); \Gamma \vdash_\Sigma N : A)\end{array}}{\Gamma \vdash_\Sigma (\lambda_{\mathcal{P}} x{:}A.M)\, N \mapsto_\beta M[N/x] : C}\ (\mathsf{O{\cdot}Red})$$

$$\frac{\begin{array}{l}\Gamma \vdash_\Sigma \lambda_{\mathcal{P}} x{:}A.M : \Pi_{\mathcal{P}} x{:}A.B \\ \Gamma \vdash_\Sigma \lambda_{\mathcal{P}} x{:}A.N : \Pi_{\mathcal{P}} x{:}A.B \quad \Gamma, x{:}A \vdash_\Sigma M \mapsto_\beta N : B\end{array}}{\Gamma \vdash_\Sigma \lambda_{\mathcal{P}} x{:}A.M \mapsto_\beta \lambda_{\mathcal{P}} x{:}A.N : \Pi_{\mathcal{P}} x{:}A.B}\ (\mathsf{O{\cdot}\lambda{\cdot}Red_1})$$

$$\frac{\begin{array}{l}\Gamma \vdash_\Sigma \lambda_{\mathcal{P}} x{:}A.M : C \\ \Gamma \vdash_\Sigma \lambda_{\mathcal{P}} x{:}B.M : C \quad \Gamma \vdash_\Sigma A \mapsto_\beta B : \mathsf{Type}\end{array}}{\Gamma \vdash_\Sigma \lambda_{\mathcal{P}} x{:}A.M \mapsto_\beta \lambda_{\mathcal{P}} x{:}B.M : C}\ (\mathsf{O{\cdot}\lambda{\cdot}Red_2})$$

$$\frac{\begin{array}{l}\Gamma \vdash_\Sigma M\, N : (\lambda_{\mathcal{P}} x{:}A.B)\, N \\ \Gamma \vdash_\Sigma P\, N : (\lambda_{\mathcal{P}} x{:}A.B)\, N \quad \Gamma \vdash_\Sigma M \mapsto_\beta P : \Pi_{\mathcal{P}} x{:}A.B\end{array}}{\Gamma \vdash_\Sigma M\, N \mapsto_\beta P\, N : (\lambda_{\mathcal{P}} x{:}A.B)\, N}\ (\mathsf{O{\cdot}Appl{\cdot}Red_1})$$

$$\frac{\begin{array}{l}\Gamma \vdash_\Sigma M\, N : (\lambda_{\mathcal{P}} x{:}A.B)\, N \\ \Gamma \vdash_\Sigma M\, P : (\lambda_{\mathcal{P}} x{:}A.B)\, N \quad \Gamma \vdash_\Sigma N \mapsto_\beta P : A\end{array}}{\Gamma \vdash_\Sigma M\, N \mapsto_\beta M\, P : (\lambda_{\mathcal{P}} x{:}A.B)\, N}\ (\mathsf{O{\cdot}Appl{\cdot}Red_2})$$

$$\frac{\Gamma \vdash_\Sigma M \mapsto_\beta N : A \quad \Gamma \vdash_\Sigma A =_\beta B : \mathsf{Type}}{\Gamma \vdash_\Sigma M \mapsto_\beta N : B}\ (\mathsf{O{\cdot}Conv{\cdot}Red})$$

Fig. 2. LF_K Reduction (Object rules)

$$\frac{\Gamma \vdash_\Sigma M{:}A \quad A \in \mathcal{A}}{\mathcal{P}(\mathcal{X}; \Gamma \vdash_\Sigma M : A)}\ (\mathsf{O{\cdot}Start_1}) \qquad \frac{\Gamma \vdash_\Sigma M{:}A}{\mathcal{P}(\emptyset; \Gamma \vdash_\Sigma M : A)}\ (\mathsf{O{\cdot}Start_2})$$

$$\frac{\mathcal{P}(\mathcal{X}; \Gamma, x{:}A \vdash_\Sigma M : B)}{\mathcal{P}(\mathcal{X} \setminus \{x\}; \Gamma \vdash_\Sigma \lambda_{\mathcal{P}} x{:}A.M : \Pi_{\mathcal{P}} x{:}A.B)}\ (\mathsf{O{\cdot}Abs})$$

$$\frac{\mathcal{P}(\mathcal{X}; \Gamma \vdash_\Sigma M : \Pi_{\mathcal{P}} x{:}A.B) \quad \mathcal{P}(\mathcal{X}; \Gamma \vdash_\Sigma N : A)}{\mathcal{P}(\mathcal{X}; \Gamma \vdash_\Sigma M\, N : (\lambda_{\mathcal{P}} x{:}A.B)\, N)}\ (\mathsf{O{\cdot}Appl})$$

$$\frac{\mathcal{P}(\mathcal{X}; \Gamma \vdash_\Sigma M : A) \quad \Gamma \vdash_\Sigma B : \mathsf{Type} \quad \Gamma \vdash_\Sigma A =_\beta B : \mathsf{Type}}{\mathcal{P}(\mathcal{X}; \Gamma \vdash_\Sigma M : B)}\ (\mathsf{O{\cdot}Conv})$$

Fig. 3. $\mathcal{P}$'s truth table

Hence, if we take $\mathcal{X} = \mathsf{Fv}(M)$, then $\mathcal{P}(\mathcal{X}; \Gamma \vdash_\Sigma M : A)$ will take into account exactly the free variables of M, according to the abovementioned intended meaning. Moreover, it is worth noticing that, once the "good families" are chosen, predicates are automatically defined as a consequence (look at the examples in the next section).

As far as definitional equality is concerned, due to lack of space, we give in Figure 4 only the rules on families, the ones for kinds and objects being similar. Notice that typing, β-reduction, and equality are mutually defined. Moreover, β-reduction is parametric over a (finite) set of good predicates, that is in LF_K we can combine several good predicates at once.

Finally, notice that our approach is different from static approaches, where "bad" terms are ruled out *a priori* via rigid type disciplines. Namely, in our framework stuck

$$\frac{\Gamma \vdash_\Sigma A : K}{\Gamma \vdash_\Sigma A =_\beta A : K}\ (\mathsf{F{\cdot}Refl{\cdot}eq}) \qquad \frac{\Gamma \vdash_\Sigma B =_\beta A : K}{\Gamma \vdash_\Sigma A =_\beta B : K}\ (\mathsf{F{\cdot}Sym{\cdot}eq})$$

$$\frac{\Gamma \vdash_\Sigma A =_\beta B : K \quad \Gamma \vdash_\Sigma B =_\beta C : K}{\Gamma \vdash_\Sigma A =_\beta C : K}\ (\mathsf{F{\cdot}Trans{\cdot}eq}) \qquad \frac{\Gamma \vdash_\Sigma A \mapsto_\beta B : K}{\Gamma \vdash_\Sigma A =_\beta B : K}\ (\mathsf{F{\cdot}Red{\cdot}eq})$$

Fig. 4. LF_κ Definitional Equality (Family rules)

$$\mathsf{A}_1 : \phi \to (\psi \to \phi) \qquad \mathsf{K} : \Box(\phi \to \psi) \to (\Box\phi \to \Box\psi)$$
$$\mathsf{A}_2 : (\phi \to (\psi \to \xi)) \to (\phi \to \psi) \to (\phi \to \xi) \qquad \mathsf{4} : \Box\phi \to \Box\Box\phi$$
$$\mathsf{A}_3 : (\neg\phi \to \neg\psi) \to ((\neg\phi \to \psi) \to \phi) \qquad \top : \Box\phi \to \phi$$
$$\mathsf{MP} : \frac{\phi \quad \phi \to \psi}{\psi} \qquad \mathsf{NEC} : \frac{\phi}{\Box\phi}$$

Fig. 5. Hilbert style rules for Modal Logic S_4

redexes can become enabled in the future. Consider, *e.g.* a redex $(\lambda_{\mathcal{P}_1} x{:}A.M)\ N$ which is stuck because a free variable y occurring into N does not satisfy the constraint imposed by predicate $\mathcal{P}_1$. Then, it could be the case that such redex is inserted into a context where y will be instantiated by a term P, by means of an outer (non-stuck) redex, like, *e.g.* in $(\lambda_{\mathcal{P}_2} y{:}B.(\lambda_{\mathcal{P}_1} x{:}A.M)\ N)\ P$. The resulting redex $(\lambda_{\mathcal{P}_1} x{:}A[P/y].M[P/y])$ $N[P/y]$ could then fire since the constraint imposed by the predicate $\mathcal{P}_1$ is satisfied by $N[P/y]$.

3 Instantiating LF_κ to Modal Logics

The Conditional Logical Framework is quite expressive. By instantiating the set of predicates, we can recover various known and new interesting Logical Frameworks. The original LF can be recovered by considering the trivial predicate induced by the set $\mathcal{A}$ of all families. More interesting instances of LF_κ are introduced below for providing smooth encodings of Modal Logics.

Modal Logic in Hilbert style. The expressive power of the Conditional Logical Framework allows to encode smoothly and uniformly both rules of proof as well as rules of derivation. We recall that the former are rules which apply only to premises which do not depend on any assumption, such as the rule of *necessitation* in Modal Logics, while the latter apply to all premises, such as *modus ponens*. The idea is to use a conditioned Π-abstraction in rules of proof and a standard Π-abstraction in rules of derivation.

We shall not develop here the encodings of all the gamut of Modal Logics, in Hilbert style, which is extensively treated in [AHMP98]. By way of example, we shall only give the signature for classical S_4 (see Figure 5) in Hilbert style (see Figure 6), which features necessitation (rule NEC in Figure 5) as a rule of proof. For notational convention in Figure 6 and in the rest of this section, we will denote by o^n the expression $\underbrace{o \to o \to \ldots \to o}_{n}$. The target language of the encoding is the instance of LF_κ, obtained by combining standard β-reduction with the β-reduction conditioned by the predicate

Closed_o induced by the set $\mathcal{A} = \{o\}$. Intuitively, $\mathsf{Closed}_o(\mathsf{Fv}(M); \Gamma \vdash_{S_4} M : \mathsf{True}(\phi))$ holds iff "all free variables occurring in M belong to a subterm which can be typed in the derivation with o". This is precisely what is needed to encode it correctly, provided o is the type of propositions. Indeed, if all the free variables of a proof term satisfy such condition, it is clear, by inspection of the typing system's object rules (see Figure 1), that there cannot be subterms of type $\mathsf{True}(\ldots)$ containing free variables. Intuitively, this corresponds to the fact that the proof of the encoded modal formula does not depend on any assumptions. The following Adequacy Theorem can be proved in the standard way, using the properties of LF_κ in Section 4.

Theorem 1 (Adequacy of the encoding of S_4 - Syntax)
Let ϵ be an encoding function (induced by the signature in Figure 6) mapping object level formulæ of S_4 into the corresponding canonical terms[1] *of LF_κ of type o. If ϕ is a propositional modal formula with propositional free variables $x_1, \ldots, x_k$, then the following judgment $\Gamma \vdash_{S_4} \epsilon(\phi) : o$ is derivable, where $\Gamma \equiv x_1{:}o, \ldots, x_k{:}o$ and each x_i is a free propositional variable in ϕ. Moreover, if we can derive in $\mathsf{LF}_\kappa \Gamma \vdash_{S_4} M : o$ where $\Gamma \equiv x_1{:}o, \ldots, x_k{:}o$ and M is a canonical form, then there exists a propositional modal formula ϕ with propositional free variables $x_1, \ldots, x_k$ such that $M \equiv \epsilon(\phi)$.*

The proof amounts to a straightforward induction on the structure of ϕ (first part) and on the structure of M (second part). After proving the adequacy of syntax, we can proceed with the more interesting theorems about the adequacy of the truth judgments.

Theorem 2 (Adequacy of the encoding of S_4 - Truth Judgment)
$\phi_1, \ldots, \phi_h \vdash_{S_4} \phi$ if and only if there exists a canonical form M such that

$$\Gamma, y_1{:}\mathsf{True}(\epsilon(\phi_1)), \ldots, y_h{:}\mathsf{True}(\epsilon(\phi_h)) \vdash_{S_4} M : \mathsf{True}(\epsilon(\phi))$$

where $\Gamma \equiv x_1{:}o, \ldots, x_k{:}o$ for each x_i free propositional variable in $\phi_1, \ldots, \phi_h, \phi$.

Classical Modal Logic S_4 and S_5 in Prawitz Style. By varying the notion of good types in the general format of LF_κ, one can immediately generate Logical Frameworks which accommodate both classical Modal Logics S_4 and S_5 in Natural Deduction style introduced by Prawitz. Figure 7 shows common and specific rules of S_4 and S_5.

We combine again standard β-reduction with a suitable notion of β-reduction conditioned by a predicate Boxed. As in the previous case such predicate can be defined by fixing a suitable notion of good type. In the case of S_4 a type is good if it is of the shape $\mathsf{True}(\Box A)$ for a suitable A or o. In the case of S_5 a type is good if it is either of the shape $\mathsf{True}(\Box A)$ or $\mathsf{True}(\neg\Box A)$ or o. Again the intended meaning is that all occurrences of free variables appear in subterms having a $\Box$-type or within a syntactic type o in the case of S_4, and a $\Box$-type or $\neg\Box$-type or within a syntactic type o in the case of S_5.

Thus, *e.g.* for S_4, the encoding of the Natural Deduction ($\Box$I) rule of Prawitz (see Figure 7) can be rendered as $\Pi\phi{:}o.\ \Pi_{\mathsf{Boxed}} x{:}\mathsf{True}(\phi).\ \mathsf{True}(\Box\phi)$, where $o{:}\mathsf{Type}$ represents formulæ, while $\mathsf{True}{:}o \to \mathsf{Type}$ and $\Box{:}o \to o$.

[1] In this case, as in [HHP93], in stating the adequacy theorem it is sufficient to consider long $\lambda\beta\eta$-normal forms without stuck redexes as canonical forms. Namely, non-reducible terms with stuck redexes must contain free variables not belonging to subterms typable with o, and clearly such terms do not correspond to any S_4-formula.

Propositional Connectives and Judgments

o : Type $\quad \supset : o^3 \quad \neg : o^2 \quad \Box : o^2 \quad \mathsf{True} : o \to \mathsf{Type}$

Propositional Axioms

$\mathsf{A}_1 \quad : \Pi\phi{:}o.\ \Pi\psi{:}o.\ \mathsf{True}(\phi \supset (\psi \supset \phi))$

$\mathsf{A}_2 \quad : \Pi\phi{:}o.\ \Pi\psi{:}o.\ \Pi\xi{:}o.\ \mathsf{True}((\phi \supset (\psi \supset \xi)) \supset (\phi \supset \psi) \supset (\phi \supset \xi))$

$\mathsf{A}_3 \quad : \Pi\phi{:}o.\ \Pi\psi{:}o.\ \mathsf{True}((\neg\psi \supset \neg\phi) \supset ((\neg\psi \supset \phi) \supset \psi))$

Modal Axioms

$\mathsf{K} \quad : \Pi\phi{:}o.\ \Pi\psi{:}o.\ \mathsf{True}(\Box(\phi \supset \psi) \supset (\Box\phi \supset \Box\psi))$

$4 \quad : \Pi\phi{:}o.\ \mathsf{True}(\Box\phi \supset \Box\Box\phi)$

$\top \quad : \Pi\phi{:}o.\ \mathsf{True}(\Box\phi \supset \phi)$

Rules

$\mathsf{MP} \quad : \Pi\phi{:}o.\ \Pi\psi{:}o.\ \mathsf{True}(\phi) \to \mathsf{True}(\phi \supset \psi) \to \mathsf{True}(\psi)$

$\mathsf{NEC} : \Pi\phi{:}o.\ \Pi_{\mathsf{Closed}_o} x{:}\mathsf{True}(\phi).\ \mathsf{True}(\Box\phi)$

Fig. 6. The signature Σ_{S_4} for classic S_4 Modal Logic in Hilbert style

Modal Logic common rules in Natural Deduction style

$$\frac{\Gamma \vdash \phi \quad \Gamma \vdash \psi}{\Gamma \vdash \phi \wedge \psi}\ (\wedge\mathsf{I}) \qquad \frac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \phi}\ (\wedge\mathsf{E}_1) \qquad \frac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \psi}\ (\wedge\mathsf{E}_2)$$

$$\frac{\Gamma \vdash \phi}{\Gamma \vdash \phi \vee \psi}\ (\vee\mathsf{I}_1) \qquad \frac{\Gamma \vdash \psi}{\Gamma \vdash \phi \vee \psi}\ (\vee\mathsf{I}_2) \qquad \frac{\Gamma \vdash \phi \vee \psi \quad \Gamma, \phi \vdash \xi \quad \Gamma, \psi \vdash \xi}{\Gamma \vdash \xi}\ (\vee\mathsf{E})$$

$$\frac{\Gamma, \phi \vdash \psi}{\Gamma \vdash \phi \to \psi}\ (\to\mathsf{I}) \qquad \frac{\Gamma \vdash \phi \to \psi \quad \Gamma \vdash \phi}{\psi}\ (\to\mathsf{E})$$

$$\frac{\Gamma, \phi \vdash \neg\phi}{\Gamma \vdash \neg\phi}\ (\neg\mathsf{I}) \qquad \frac{\Gamma \vdash \neg\phi \quad \Gamma \vdash \phi}{\Gamma \vdash \psi}\ (\neg\mathsf{E}) \qquad \frac{\Gamma, \neg\phi \vdash \phi}{\Gamma \vdash \phi}\ (\mathsf{RAA})$$

Specific rules for Modal Logic S_4 in Natural Deduction style

$$\frac{\Box\Gamma \vdash \phi}{\Box\Gamma \vdash \Box\phi}\ (\Box\mathsf{I}) \qquad \frac{\Gamma \vdash \Box\phi}{\Gamma \vdash \phi}\ (\Box\mathsf{E})$$

Specific rules for Modal Logic S_5 in Natural Deduction style

$$\frac{\Box\Gamma_0, \neg\Box\Gamma_1 \vdash \phi}{\Box\Gamma_0, \neg\Box\Gamma_1 \vdash \Box\phi}\ (\Box\mathsf{I}) \qquad \frac{\Gamma \vdash \Box\phi}{\Gamma \vdash \phi}\ (\Box\mathsf{E})$$

Fig. 7. Modal Logic (common rules and $S_{4,5}$ rules) in LF_K

Quite a remarkable property of this signature is that it encodes a slightly more usable version of Natural Deduction S_4 than the one originally introduced by Prawitz. Our formulation is precisely what is needed to achieve a normalization result in the logic which could not be done in the original system of Prawitz. Being able to refer to *boxed subterms*, rather than just boxed variables, is what *makes the difference*. Once again LF_K encodings *improve presentations of logical systems*!

Propositional Connectives and Judgments

o : Type and : o^3 or : o^3 $\supset$: o^3 $\neg$: o^2 $\Box$: o^2 True : $o \to$ Type

Propositional Rules

$\mathsf{And_I}$: $\Pi\phi{:}o.\ \Pi\psi{:}o.\ \mathsf{True}(\phi) \to \mathsf{True}(\psi) \to \mathsf{True}(\phi\ \mathsf{and}\ \psi)$

$\mathsf{And_{E_1}}$: $\Pi\phi{:}o.\ \Pi\psi{:}o.\ \mathsf{True}(\phi\ \mathsf{and}\ \psi) \to \mathsf{True}(\phi)$

$\mathsf{And_{E_2}}$: $\Pi\phi{:}o.\ \Pi\psi{:}o.\ \mathsf{True}(\phi\ \mathsf{and}\ \psi) \to \mathsf{True}(\psi)$

$\mathsf{Or_{I_1}}$: $\Pi\phi{:}o.\ \Pi\psi{:}o.\ \mathsf{True}(\phi) \to \mathsf{True}(\phi\ \mathsf{or}\ \psi)$

$\mathsf{Or_{I_2}}$: $\Pi\phi{:}o.\ \Pi\psi{:}o.\ \mathsf{True}(\psi) \to \mathsf{True}(\phi\ \mathsf{or}\ \psi)$

$\mathsf{Or_E}$: $\Pi\phi{:}o.\ \Pi\psi{:}o.\ \mathsf{True}(\phi\ \mathsf{or}\ \psi) \to (\mathsf{True}(\phi) \to \mathsf{True}(\xi)) \to (\mathsf{True}(\psi) \to \mathsf{True}(\xi)) \to \mathsf{True}(\xi)$

$\mathsf{Imp_I}$: $\Pi\phi{:}o.\ \Pi\psi{:}o.\ (\mathsf{True}(\phi) \to \mathsf{True}(\psi)) \to \mathsf{True}(\phi \supset \psi)$

$\mathsf{Imp_E}$: $\Pi\phi{:}o.\ \Pi\psi{:}o.\ \mathsf{True}(\phi \supset \psi) \to \mathsf{True}(\phi) \to \mathsf{True}(\psi)$

$\mathsf{Neg_I}$: $\Pi\phi{:}o.\ (\mathsf{True}(\phi) \to \mathsf{True}(\neg\phi)) \to \mathsf{True}(\neg\phi)$

$\mathsf{Neg_E}$: $\Pi\phi{:}o.\ \Pi\psi{:}o.\ \mathsf{True}(\neg\phi) \to \mathsf{True}(\phi) \to \mathsf{True}(\psi)$

RAA : $\Pi\phi{:}o.\ (\mathsf{True}(\neg\phi) \to \mathsf{True}(\phi)) \to \mathsf{True}(\phi)$

Modal Rules

$\mathsf{Box_I}$: $\Pi\phi{:}o.\ \Pi_{\mathsf{Boxed}\,x}{:}\mathsf{True}(\phi).\ \mathsf{True}(\Box\phi)$

$\mathsf{Box_E}$: $\Pi\phi{:}o.\ \Pi x{:}\mathsf{True}(\Box\phi).\ \mathsf{True}(\phi)$

Fig. 8. The signature Σ_S for classic S_4 or S_5 Modal Logic in Natural Deduction style

4 Properties of $\mathsf{LF_K}$

In this section, we study relevant properties of $\mathsf{LF_K}$. We show that, without any extra assumption on the predicates, the type system satisfies a list of basic properties, including the subderivation property, subject reduction and strong normalization. The latter follows easily from the strong normalization result for LF, see [HHP93]. Confluence and judgment decidability can be proved under the assumption that the various predicate reductions nicely combine, in the sense that no reduction can prevent a redex, which could fire, from firing after the reduction. The difficulty in proving subject reduction and confluence for $\mathsf{LF_K}$ lies in the fact that predicate β-reductions do not have corresponding untyped reductions, while standard proofs of subject reduction and confluence for dependent type systems are based on underlying untyped β-reductions (see *e.g.* [HHP93]). We provide an original technique, based solely on typed β-reductions, providing a fine analysis of the structure of terms which are β-equivalent to Π-terms.

In the following, we will denote by $\Gamma \vdash_\Sigma \alpha$ any judgment defined in $\mathsf{LF_K}$. The proof of the following theorem is straightforward.

Theorem 3 (Basic Properties)

Subderivation Property

1. *Any derivation of* $\Gamma \vdash_\Sigma \alpha$ *has subderivations of* Σ sig *and* $\vdash_\Sigma \Gamma$.
2. *Any derivation of* $\Sigma, a{:}K$ sig *has subderivations of* Σ sig *and* $\vdash_\Sigma K$.
3. *Any derivation of* $\Sigma, f{:}A$ sig *has subderivations of* Σ sig *and* $\vdash_\Sigma A$: Type.
4. *Any derivation of* $\vdash_\Sigma \Gamma, x{:}A$ *has subderivations of* Σ sig *and* $\Gamma \vdash_\Sigma A$: Type.

5. *Given a derivation of $\Gamma \vdash_\Sigma \alpha$ and any subterm occurring in the subject of the judgment, there exists a derivation of a smaller length of a judgment having that subterm as a subject.*
6. *If $\Gamma \vdash_\Sigma A : K$, then $\Gamma \vdash_\Sigma K$.*
7. *If $\Gamma \vdash_\Sigma M : A$, then $\Gamma \vdash_\Sigma A : \mathsf{Type}$ if there are no stuck redexes in A.*

Derivability of Weakening and Permutation
If Γ and Δ are valid contexts, and every declaration occurring in Γ also occurs in Δ, then $\Gamma \vdash_\Sigma \alpha$ implies $\Delta \vdash_\Sigma \alpha$.

Transitivity
If $\Gamma, x{:}A, \Delta \vdash_\Sigma \alpha$ and $\Gamma \vdash_\Sigma M : A$, then $\Gamma, \Delta[M/x] \vdash_\Sigma \alpha[M/x]$.

Convertibility of types in domains
1. *For all $\Gamma, x{:}A, \Delta \vdash_\Sigma \alpha$ and $\Gamma, \Delta \vdash_\Sigma A =_\beta A' : K$, then $\Gamma, x{:}A', \Delta \vdash_\Sigma \alpha$.*
2. *If $\mathcal{P}(\mathcal{X}; \Gamma, x{:}A, \Delta \vdash_\Sigma M : B)$ holds and $\Gamma, \Delta \vdash_\Sigma A =_\beta A' : K$, then $\mathcal{P}(\mathcal{X}; \Gamma, x{:}A', \Delta \vdash_\Sigma M : B)$ holds.*

Strong normalization of LF_K follows from the one of LF, since there is a trivial map of LF_K in LF, which simply forgets about predicates. Thus, if there would be an infinite reduction in LF_K, this would be mapped into an infinite reduction in LF.

Theorem 4 (Strong Normalization)
1. *If $\Gamma \vdash_\Sigma K$, then $K \in \mathsf{SN}^{\mathcal{K}}$.*
2. *if $\Gamma \vdash_\Sigma A : K$, then $A \in \mathsf{SN}^{\mathcal{F}}$.*
3. *if $\Gamma \vdash_\Sigma M : A$, then $M \in \mathsf{SN}^{\mathcal{O}}$.*

Where $\mathsf{SN}^{\{\mathcal{K},\mathcal{F},\mathcal{O}\}}$ denotes the set of strongly normalizing terms of kinds, families, and objects, respectively.

In the following we will denote by $\Gamma \vdash_\Sigma A \rightleftharpoons_\beta B : K$ the fact that either $\Gamma \vdash_\Sigma A \mapsto_\beta B : K$ or $\Gamma \vdash_\Sigma B \mapsto_\beta A : K$ holds. Moreover, in the next results we will use a *measure* of the complexity of the proofs of judgments which takes into account all the rules applied in the derivation tree. More precisely, we have the following definition:

Definition 3 (Measure of a derivation)
Given a proof $\mathcal{D}$ of the judgment $\Gamma \vdash_\Sigma \alpha$, we define the measure of $\mathcal{D}$, denoted by $\#\mathcal{D}$, as the number of all the rules applied in the derivation of $\mathcal{D}$ itself.

The following lemma is easily proved by induction on $\#\mathcal{D}$.

Lemma 2 (Reduction/Expansion)
For any derivation $\mathcal{D} : \Gamma \vdash_\Sigma A =_\beta B : K$, either $A \equiv B$ or there exist $C_1, \ldots, C_n$ ($n \geq 0$) such that:

1. *There exist $\mathcal{D}_1 : \Gamma \vdash_\Sigma A \rightleftharpoons_\beta C_1 : K$ and $\mathcal{D}_2 : \Gamma \vdash_\Sigma C_1 \rightleftharpoons_\beta C_2 : K \ldots$ and $\mathcal{D}_n : \Gamma \vdash_\Sigma C_{n-1} \rightleftharpoons_\beta C_n : K$ and $\mathcal{D}_{n+1} : \Gamma \vdash_\Sigma C_n \rightleftharpoons_\beta B : K$ and, for all $1 \leq i \leq n+1$, we have $\#\mathcal{D}_i < \#\mathcal{D}$.*
2. *For any $1 \leq i \leq n$, we have that there exist $\mathcal{D}'_1 : \Gamma \vdash_\Sigma A =_\beta C_i : K$ and $\mathcal{D}'_2 : \Gamma \vdash_\Sigma C_i =_\beta B : K$ and $\#\mathcal{D}'_1, \#\mathcal{D}'_2 < \#\mathcal{D}$.*

This lemma allows us to recover the structure of a term which is β-equivalent to a Π-term. The proof proceeds by induction on $\#\mathcal{D}$.

Lemma 3 (Key lemma)

1. *If* $\mathcal{D} : \Gamma \vdash_\Sigma \Pi_\mathcal{P} x{:}A.K =_\beta K'$ *holds, then either* $\Pi_\mathcal{P} x{:}A.K \equiv K'$ *or there are* $\mathcal{P}_1, ..., \mathcal{P}_n$, *and* $D_1, ..., D_n$, *and* $M_1, ..., M_n$ ($n \geq 0$), *and* K_A, $\mathcal{D}_1$, $\mathcal{D}_2$ *such that:*
   (a) $K' \equiv ((\lambda_{\mathcal{P}_1} y_1{:}D_1. \ldots ((\lambda_{\mathcal{P}_n} y_n{:}D_n.(\Pi_\mathcal{P} x{:}A'.K'')) M_n) \ldots) M_1)$.
   (b) $\mathcal{D}_1 : \Gamma \vdash_\Sigma A =_\beta ((\lambda_{\mathcal{P}_1} y_1{:}D_1. \ldots ((\lambda_{\mathcal{P}_n} y_n{:}D_n.A') M_n) \ldots) M_1) : K_A$.
   (c) $\mathcal{D}_2 : \Gamma, x{:}A \vdash_\Sigma K =_\beta ((\lambda_{\mathcal{P}_1} y_1{:}D_1. \ldots ((\lambda_{\mathcal{P}_n} y_n{:}D_n.K'') M_n) \ldots) M_1)$.
   (d) $\#\mathcal{D}_1, \#\mathcal{D}_2 < \#\mathcal{D}$.
2. *If* $\mathcal{D} : \Gamma \vdash_\Sigma \Pi_\mathcal{P} x{:}A.B =_\beta C : K$ *holds, then either* $\Pi_\mathcal{P} x{:}A.B \equiv C$ *or there are* $\mathcal{P}_1, ..., \mathcal{P}_n$, *and* $D_1, ..., D_n$, *and* $M_1, ..., M_n$ ($n \geq 0$), *and* K_A, K_B, *and* $\mathcal{D}_1$, $\mathcal{D}_2$ *such that:*
   (a) $C \equiv ((\lambda_{\mathcal{P}_1} y_1{:}D_1. \ldots ((\lambda_{\mathcal{P}_n} y_n{:}D_n.(\Pi_\mathcal{P} x{:}A'.B')) M_n) \ldots) M_1)$.
   (b) $\mathcal{D}_1 : \Gamma \vdash_\Sigma A =_\beta ((\lambda_{\mathcal{P}_1} y_1{:}D_1. \ldots ((\lambda_{\mathcal{P}_n} y_n{:}D_n.A') M_n) \ldots) M_1) : K_A$.
   (c) $\mathcal{D}_2 : \Gamma, x{:}A \vdash_\Sigma B =_\beta ((\lambda_{\mathcal{P}_1} y_1{:}D_1. \ldots ((\lambda_{\mathcal{P}_n} y_n{:}D_n.B') M_n) \ldots) M_1) : K_B$.
   (d) $\#\mathcal{D}_1, \#\mathcal{D}_2 < \#\mathcal{D}$.

Corollary 1 (Π's injectivity)

1. *If* $\Gamma \vdash_\Sigma \Pi_\mathcal{P} x{:}A.K =_\beta \Pi_\mathcal{P} x{:}A'.K'$, *then* $\Gamma \vdash_\Sigma A =_\beta A' : K_A$ *and* $\Gamma, x{:}A \vdash_\Sigma K =_\beta K'$.
2. *If* $\Gamma \vdash_\Sigma \Pi_\mathcal{P} x{:}A.B =_\beta \Pi_\mathcal{P} x{:}A'.B' : K$, *then* $\Gamma \vdash_\Sigma A =_\beta A' : K'$ *and* $\Gamma, x{:}A \vdash_\Sigma B =_\beta B' : K''$.

The proof of the following theorem uses the Key Lemma.

Theorem 5 (Unicity, Abstraction and Subject Reduction)

Unicity of Types and Kinds

1. *If* $\Gamma \vdash_\Sigma A : K_1$ *and* $\Gamma \vdash_\Sigma A : K_2$, *then* $\Gamma \vdash_\Sigma K_1 =_\beta K_2$.
2. *If* $\Gamma \vdash_\Sigma M : A_1$ *and* $\Gamma \vdash_\Sigma M : A_2$, *then* $\Gamma \vdash_\Sigma A_1 =_\beta A_2 : K$.

Abstraction Typing

1. *If* $\Gamma \vdash_\Sigma \lambda_\mathcal{P} x{:}A.T : \Pi_\mathcal{P} x{:}A'.T'$, *then* $\Gamma \vdash_\Sigma A =_\beta A' : K$.
2. *If* $\Gamma \vdash_\Sigma \lambda_\mathcal{P} x{:}A.T : \Pi_\mathcal{P} x{:}A.T'$, *then* $\Gamma, x{:}A \vdash_\Sigma T : T'$.

Subject Reduction

1. *If* $\Gamma \vdash_\Sigma (\lambda_\mathcal{P} x{:}A.K)\, N$, *then* $\Gamma \vdash_\Sigma K[N/x]$.
2. *If* $\Gamma \vdash_\Sigma (\lambda_\mathcal{P} x{:}A.B)\, N : K$ *and* $\mathcal{P}(\mathsf{Fv}(N); \Gamma \vdash_\Sigma N : A)$ *holds, then* $\Gamma \vdash_\Sigma B[N/x] : K$.
3. *If* $\Gamma \vdash_\Sigma (\lambda_\mathcal{P} x{:}A.M)\, N : C$ *and* $\mathcal{P}(\mathsf{Fv}(N); \Gamma \vdash_\Sigma N : A)$ *holds, then* $\Gamma \vdash_\Sigma M[N/x] : C$.

In the following, we consider notions of reduction for LF_κ that are *well-behaved* in the following sense:

1. a redex which can fire, can still fire after any β-reduction in its argument (possibly corresponding to a different predicate);
2. a redex which can fire, can still fire after application to its argument of a substitution coming from another reduction.

Formally:

Definition 4 (Well behaved β-reduction)
Assume that the LF_κ *β-reduction is determined by the set* $\mathbf{P}$ *of good predicates. Then the β-reduction is* well-behaved *if, for all* $\mathcal{P}, \mathcal{P}' \in \mathbf{P}$*, the following two conditions are satisfied:*

1. *If* $\mathcal{P}(\mathsf{Fv}(N); \Gamma \vdash_\Sigma N : A)$ *holds and* $\Gamma \vdash_\Sigma N \mapsto_\beta N' : A$*, then* $\mathcal{P}(\mathsf{Fv}(N'); \Gamma \vdash_\Sigma N' : A)$ *holds.*
2. *If* $\mathcal{P}(\mathsf{Fv}(N); \Gamma', y{:}A'; \Gamma \vdash_\Sigma N : A)$ *and* $\mathcal{P}'(\mathsf{Fv}(N'); \Gamma' \vdash_\Sigma N' : A')$ *hold, then* $\mathcal{P}(\mathsf{Fv}(N[N'/y]); \Gamma', \Gamma[N'/y] \vdash_\Sigma N[N'/y] : A[N'/y])$ *holds.*

Definition 4 above allows one to combine several notions of predicate reduction, provided the latter are all well-behaved.

Since LF_κ is strongly normalizing, in order to prove confluence of the system, by *Newman's Lemma*, it is sufficient to show that LF_κ β-reduction is *locally confluent*, *i.e.* (in the case of objects) if $\Gamma \vdash_\Sigma M_1 \mapsto_\beta M_2 : C$ and $\Gamma \vdash_\Sigma M_1 \mapsto_\beta M_3 : C$, then there exists M_4 such that $\Gamma \vdash_\Sigma M_2 \mapsto\!\!\!\!\to_\beta M_4 : C$ and $\Gamma \vdash_\Sigma M_3 \mapsto\!\!\!\!\to_\beta M_4 : C$. Under the hypothesis that β-reduction is well-behaved, using Theorem 5, we can prove that the reduction is locally confluent.

Theorem 6 (Local Confluence)
If β-reduction is well behaved, then it is locally confluent.

Finally, from Newman's Lemma, using Theorems 4 and 6, we have:

Theorem 7 (Confluence)
Assume β-reduction is well behaved. Then the relation $\mapsto_\beta$ is confluent, i.e.:

1. *If* $\Gamma \vdash_\Sigma K_1 \mapsto\!\!\!\!\to_\beta K_2$ *and* $\Gamma \vdash_\Sigma K_1 \mapsto\!\!\!\!\to_\beta K_3$*, then there exists* K_4 *such that* $\Gamma \vdash_\Sigma K_2 \mapsto\!\!\!\!\to_\beta K_4$ *and* $\Gamma \vdash_\Sigma K_3 \mapsto\!\!\!\!\to_\beta K_4$*.*
2. *If* $\Gamma \vdash_\Sigma A_1 \mapsto\!\!\!\!\to_\beta A_2 : K$ *and* $\Gamma \vdash_\Sigma A_1 \mapsto\!\!\!\!\to_\beta A_3 : K$*, then there exists* A_4 *such that* $\Gamma \vdash_\Sigma A_2 \mapsto\!\!\!\!\to_\beta A_4 : K$ *and* $\Gamma \vdash_\Sigma A_3 \mapsto\!\!\!\!\to_\beta A_4 : K$*.*
3. *If* $\Gamma \vdash_\Sigma M_1 \mapsto\!\!\!\!\to_\beta M_2 : C$ *and* $\Gamma \vdash_\Sigma M_1 \mapsto\!\!\!\!\to_\beta M_3 : C$*, then there exists* M_4 *such that* $\Gamma \vdash_\Sigma M_2 \mapsto\!\!\!\!\to_\beta M_4 : C$ *and* $\Gamma \vdash_\Sigma M_3 \mapsto\!\!\!\!\to_\beta M_4 : C$*.*

Judgements decidability show that LF_κ can be used as a framework for proof checking.

Theorem 8 (Judgements decidability of LF_κ)
If $\mapsto_\beta$ is well-behaved, then it is decidable whether $\Gamma \vdash_\Sigma \alpha$ is derivable.

The standard pattern of the proof applies, provided we take care that reductions are typed in computing the normal form of a type.

It is easy to show that, for all instances of LF_κ considered in Section 3, the corresponding β-reductions are well behaved, thus judgement decidability holds.

5 Conclusions and Directions for Future Work

In this paper, we have investigated the language theory of the Conditional Logical Framework LF_κ, which subsumes the Logical Framework LF of [HHP93], and generates new Logical Frameworks. These can feature a very broad spectrum of generalized typed (possibly by value) β-reductions, together with an expressive type system

which records when such reductions have not yet fired. The key ingredient in the typing system is a decomposition of the standard term-application rule. A very interesting feature of our system is that it allows for dealing with values induced by the typing system, *i.e.* values which are determined by the typing system, through the notion of good predicates. We feel that our investigation of LF_κ is quite satisfactory: we have proved major metatheoretical results, such as strong normalization, subject reduction and confluence (this latter under a suitable assumption). For LF_κ we have achieved decidability, which legitimates it as a metalanguage for proof checking and interactive proof editing. We have shown how suitable instances of LF_κ provide smooth encodings of Modal Logics, compared with the heavy machinery needed when we work directly into LF, see *e.g.* [AHMP98]. Namely, the work of specifying the variable occurrence side-conditions is factored out once and for all into the framework.

Here is a list of comments and directions for future work.

- Some future efforts should be devoted to the task of investigating the structure of canonical forms including stuck redexes. Such analysis could clarify the rôle of stuck β-reductions and stuck terms in the activity of encoding object logics into LF_κ. Moreover, following the approach carried out in [WCPW02], we could benefit from a presentation of LF_κ based upon a clear characterization of canonical forms in order to avoid the notion of β-conversion and the related issues.
- We believe that our metalogical Framework has some considerable potential. In particular, it could be useful for modeling dynamic situations, where the static approach of rigid typed disciplines is not sufficient. We plan to carry out more experiments in the future, *e.g.* in the field of reactive systems, where the rôle of stuck redexes could be very helpful in modeling the dynamics of variables instantiations.
- Our results should scale up to systems corresponding to the full Calculus of Constructions [CH88].
- Is there an interesting Curry-Howard isomorphism for LF_κ, and for other systems blending rewriting facilities and higher order calculi?
- Investigate whether LF_κ could give sharp encodings of Relevance and Linear Logics. Is the notion of good predicate involved in the definition of LF_κ useful in this respect? Or do we need a different one?
- Compare with work on Deduction Modulo [DHK03].
- In [KKR90], Kirchner-Kirchner-Rusinowitch developed an *Algebraic Logical Framework* for first-order constrained deduction. Deduction rules and constraints are given for a first-order logic with equality. Enhancing LF_κ with constraints seems to be a perfect fit for a new race of metalanguages for proof checking and automatic theorem proving. Without going much into the details of our future research, the abstraction-term could, indeed, have the shape $\lambda_{\mathcal{P}}\overline{x};\mathcal{C}.M$, where $\mathcal{P}$ records the first-order formula, $\overline{x}$ is a vector of variables occurring in the formula and $\mathcal{C}$ are constraints over $\overline{x}$.
- Until now, the predicate states a condition that takes as *input* the argument and its type. It would be interesting to extend the framework with another predicate, say $\mathcal{Q}$, applied to the body of the function. The abstraction would then have the

form $\lambda_{\mathcal{P}} x{:}A.M^{\mathcal{Q}}$. This extension would put conditions on the function *output*, so leading naturally to a framework for defining Program Logics *à la* Hoare-Floyd.
– Implement new proof assistants based on dependent type systems, like *e.g.* Coq, based on LF_{κ}.

References

[AHMP98] Avron, A., Honsell, F., Miculan, M., Paravano, C.: Encoding Modal Logics in Logical Frameworks. Studia Logica 60(1), 161–208 (1998)

[BCKL03] Barthe, G., Cirstea, H., Kirchner, C., Liquori, L.: Pure Pattern Type Systems. In: Proc. of POPL, pp. 250–261. ACM Press, New York (2003)

[CH84] Cresswell, M., Hughes, G.: A companion to Modal Logic. Methuen (1984)

[CH88] Coquand, T., Huet, G.: The Calculus of Constructions. Information and Computation 76(2/3), 95–120 (1988)

[CKL01a] Cirstea, H., Kirchner, C., Liquori, L.: Matching Power. In: Middeldorp, A. (ed.) RTA 2001. LNCS, vol. 2051, pp. 77–92. Springer, Heidelberg (2001)

[CKL01b] Cirstea, H., Kirchner, C., Liquori, L.: The Rho Cube. In: Honsell, F., Miculan, M. (eds.) FOSSACS 2001. LNCS, vol. 2030, pp. 166–180. Springer, Heidelberg (2001)

[DHK03] Dowek, G., Hardin, T., Kirchner, C.: Theorem Proving Modulo. Journal of Automated Reasoning 31(1), 33–72 (2003)

[HHP93] Harper, R., Honsell, F., Plotkin, G.: A Framework for Defining Logics. Journal of the ACM 40(1), 143–184 (1993); Preliminary version in proc. of LICS 1987

[HLL07] Honsell, F., Lenisa, M., Liquori, L.: A Framework for Defining Logical Frameworks. Computation, Meaning and Logic. Articles dedicated to Gordon Plotkin, Electr. Notes Theor. Comput. Sci. 172, 399–436 (2007)

[KKR90] Kirchner, C., Kirchner, H., Rusinowitch, M.: Deduction with Symbolic Constraints. Technical Report 1358, INRIA, Unité de recherche de Lorraine, Vandoeuvre-lès-Nancy, FRANCE (1990)

[NPP08] Nanevski, A., Pfenning, F., Pientka, B.: Contextual Model Type Theory. ACM Transactions on Computational Logic 9(3) (2008)

[WCPW02] Watkins, K., Cervesato, I., Pfenning, F., Walker, D.: A Concurrent Logical Framework I: Judgments and Properties. Technical Report CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University (2002)

Nominal Renaming Sets

Murdoch J. Gabbay[1] and Martin Hofmann[2]

[1] http://www.gabbay.org.uk
[2] http://www.tcs.informatik.uni-muenchen.de/~mhofmann/

Abstract. Nominal techniques are based on the idea of sets with a finitely-supported atoms-permutation action.

We consider the idea of *nominal renaming sets*, which are sets with a finitely-supported atoms-renaming action; renamings can identify atoms, permutations cannot. We show that nominal renaming sets exhibit many of the useful qualities found in (permutative) nominal sets; an elementary sets-based presentation, inductive datatypes of syntax up to binding, cartesian closure, and being a topos. Unlike is the case for nominal sets, the notion of names-abstraction coincides with functional abstraction. Thus we obtain a concrete presentation of sheaves on the category of finite sets in the form of a category of sets with structure.

Keywords: Nominal renaming sets, nominal sets, abstract syntax with binding.

1 Introduction

The motivation of this work is to provide semantic foundations for formal theories of syntax with variable binding. Several such theories have been proposed in the literature [5,17,16,14,19] and used in concrete applications with varying success [15,4,25,18,21]. All but the most elementary approaches require some semantical or proof-theoretical justification asserting their soundness and also explaining the meaning of the judgements made. Functor categories (categories of (pre)sheaves) are a popular method [7,16,3,8] for providing such foundations. However, proofs involving presheaves are notoriously complicated and require a thorough working knowledge so as to be able to reduce clutter by elision of trivial steps. In some situations [21,25], a sets-based semantics is available in the form of nominal sets [14] which considerably simplifies metatheoretic reasoning. A domains version of nominal sets [20] has also become an important tool in denotational semantics of languages with dynamic allocation [1]

This paper provides a sets-based foundation for another class of systems for higher-order syntax including in particular the *theory of contexts* [17] which could hitherto only be modelled with functor categories. This will allow us to considerably simplify the cumbersome functor-theoretic proof of soundness of that theory [3]. In this way, we expect that our sets-based presentation of the semantics will then allow to justify further extensions which would until now be too complicated to work through.

Note that we do not propose yet another approach to higher-order syntax. Rather, we introduce a simplified presentation, up to categorical equivalence, of an existing semantic model. We now summarise the contributions of the paper in more detail.

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 158–173, 2008.

Write $\mathbb{I}$ for the category of finite sets and injections between them, write $\mathbb{F}$ for the category of finite sets and all (not necessarily injective) functions between them, and write Set for the category of sets[1] and functions between them.

Previous work presented nominal sets [13]. These can be presented as the category of pullback-preserving presheaves in $\mathsf{Set}^{\mathbb{I}}$. Simultaneously, Fiore et al [7] and Hofmann [16] proposed to use $\mathsf{Set}^{\mathbb{F}}$ for higher-order abstract syntax. Both can be used as mathematical models for inductive specification and reasoning on syntax with binding.

In $\mathsf{Set}^{\mathbb{F}}$ for a presheaf F, the presheaf F^+ given by $F^+(X) = F(X \cup \{x\})$ for $x \notin X$ is isomorphic to the exponential $\mathbf{A} \Rightarrow F$. In the category of nominal sets the corresponding presheaf is isomorphic to $\mathbf{A} \multimap F$, with $\multimap$ being the right adjoint to a tensor product different from cartesian product (it is mentioned in Theorem 34 and is written $[\mathbf{A}]\mathbf{X}$). In either case the presheaf $\mathbf{A}$ is given by $\mathbf{A}(S) = S$. Thus, $\mathsf{Set}^{\mathbb{F}}$ seems better-suited for modelling higher-order abstract syntax, which uses typings like

$$\mathit{lam} : (\mathit{var} \to \mathit{tm}) \to \mathit{tm} \tag{1}$$

for variable binding constructs (in this case: 'lambda'). This arises when using an existing theorem prover or type theory (for example Coq) to model higher-order abstract syntax [6,17]. In particular, in [3] the presheaf category from [16] was used to establish soundness of the theory of contexts from [17].

FreshML [23] and the deep embeddings of the nominal datatypes package [25] provide a syntactic primitive corresponding to the right adjoint $\multimap$ mentioned above. They use typings like

$$\mathit{lam} : (\mathit{var} \multimap \mathit{tm}) \to \mathit{tm}. \tag{2}$$

Precomposing with the canonical map $(A \to B) \longrightarrow (A \multimap B)$, one could justify the typing in (1), but *lam* would not be injective. For the typing

$$\mathit{lam} : (\mathit{tm} \to \mathit{tm}) \to \mathit{tm} \tag{3}$$

used in the Twelf system [19], another presheaf topos ($\hat{\mathbb{S}}$ in [16]) is adequate. The constructions of this paper cannot be applied to this directly, but see the Conclusions.

The nominal approach benefits from a *sets-based* presentation. Denotations of types are sets with structure — whence the name 'nominal *sets*' — rather than functors.

In this paper we offer a sets-based presentation of the full subcategory of $\mathsf{Set}^{\mathbb{F}}$ of functors preserving pullbacks of monos. We present it as a category of sets with a not-necessarily-injective *renaming action*. The pullback requirement implies that $F(X \cap Y)$ is isomorphic to 'the intersection of' $F(X)$ and $F(Y)$, in line with the intuition 'objects with free variables from X' for $F(X)$. This rules out artefacts present in $\mathsf{Set}^{\mathbb{F}}$ like $F(X) = \mathit{if}\ |X| > 1\ \mathit{then}\ \{\star\}\ \mathit{else}\ \emptyset$, which never arise as denotations of types in higher-order abstract syntax.

In [16] it was argued that the internal logic of $\mathsf{Set}^{\mathbb{F}}$ qua topos might be unsuited to reasoning with higher-order abstract syntax since equality of atoms (names, variables) is not decidable in it. It was proposed in *loc. cit.* to import the logic of the Schanuel topos

[1] It is convenient, but not necessary, to take this to be the category of 'ordinary' ZF sets (not mentioning atoms). Since we are working at the meta-level, this is in fact not important and any sufficiently rich collection of collections of elements will do.

using a pullback of triposes. Our presentation of this pullback construction amounts to interpreting predicates as *equivariant subsets*; subsets of a renaming set stable under injective renamings. Staton and Fiore [8] define a category of substitution sets equivalent to a sheaf subcategory of $\mathsf{Set}^{\mathbb{F}}$. Their category is not defined concretely in terms of ordinary sets but as a theory within the Schanuel topos. It does not make $\mathsf{Set}^{\mathbb{F}}$ easier to work with (and Staton and Fiore never introduced it for that purpose).

We also characterise the function space of nominal renaming sets as a set of functions, rather than a Kripke exponential transported along the equivalence, and we identify the tripos structure needed to reason about renaming sets in a robust way, and in particular, to justify the theory of contexts.

Some notation: This paper uses three different kinds of arrow. For the reader's convenience we survey their definitions; this is all standard.

– If X and Y are sets then $X \to Y$ is the set of functions from X to Y.

– If $\mathbf{X}$ and $\mathbf{Y}$ are nominal renaming sets then $\mathbf{X} \longrightarrow \mathbf{Y}$ is the set of arrows from $\mathbf{X}$ to $\mathbf{Y}$ in the category Ren (Definition 5). These are maps in $|\mathbf{X}| \to |\mathbf{Y}|$ with empty support (Lemma 25).

– $\mathbf{X} \Rightarrow \mathbf{Y}$ is the exponential (Definition 20 and Theorem 26). These are maps in $|\mathbf{X}| \to |\mathbf{Y}|$ with finite support (Lemma 21).

2 Nominal Renaming Sets

Definition 1. *Fix a countably infinite* set of atoms $\mathbf{A}$. *We assume that atoms are disjoint from numbers* $0, 1, 2, \ldots$, *truth-values* $\bot, \top$, *and other standard mathematical entities.* $\mathbf{A}$ *can be viewed as a set of* urelemente *[2,13,10]; we view them as a model of* names *or* variable symbols.

$a, b, c, \ldots$ *will range over atoms. We follow a* permutative convention *that simultaneously introduced metavariables for atoms range* permutatively*; for example* a *and* b *range over any two* distinct *atoms.*[2]

Definition 2. *Let* Fin *be the set of functions* $\sigma \in \mathbf{A} \to \mathbf{A}$ *such that there exists some finite* $S \subseteq \mathbf{A}$ *such that for all* $b \in \mathbf{A} \setminus S$ *it is the case that* $\sigma(b) = b$.

σ, τ will range over elements of Fin. We call these *(finitely supported) renamings.*

Definition 3. *Write* $[a_1 \mapsto y_1, \ldots, a_k \mapsto y_k]$ *for the function that maps* a_i *to* y_i *for* $1 \leq i \leq k$ *and maps all other* b *(that is, atoms* b *not in the set* $\{a_1, \ldots, a_k\}$*) to themselves. Note that every function in* Fin *can be written in this fashion.*

In particular, write $[a \mapsto b]$ *for the function which 'maps* a *to* b*':*

$$[a \mapsto b](a) = b \qquad [a \mapsto b](b) = b \quad \textit{and} \quad [a \mapsto b](c) = c$$

We write $\circ$ *for functional composition. For example* $[a \mapsto b] \circ [b \mapsto a] = [a \mapsto b]$ *(and* $[a \mapsto b] \circ [b \mapsto a] \neq [a \mapsto b, b \mapsto a]$*). Write* id *for the* identity renaming. $id(a) = a$ *always.*

[2] Note that we are working in the usual naïve set-theoretic metalanguage; atoms form a set. At that level, atoms have a decidable equality.

Fin with $\circ$ and id is a monoid. That is, $id \circ \sigma = \sigma \circ id = \sigma$ and $\sigma \circ (\sigma' \circ \sigma'') = (\sigma \circ \sigma') \circ \sigma''$. If $S \subseteq \mathbf{A}$ and $\sigma \in Fin$, write $\sigma|_S$ for the partial function equal to σ on S and undefined elsewhere.

Definition 4. *A* nominal renaming set $\mathbf{X}$ *is a pair* $(|\mathbf{X}|, \cdot)$ *of an* underlying set $|\mathbf{X}|$ *and a* finitely-supported renaming action $\cdot$.

A finitely-supported renaming action $\cdot$ *is a map from* $Fin \times |\mathbf{X}|$ *to* $|\mathbf{X}|$ *such that:*

– $id \cdot x = x$ *always.*

– $\sigma' \cdot (\sigma \cdot x) = (\sigma' \circ \sigma) \cdot x$ *always.*

– *For every* $x \in |\mathbf{X}|$ *there is a finite* $S \subseteq \mathbf{A}$ *such that* $\sigma|_S = \sigma'|_S$ *implies* $\sigma \cdot x = \sigma' \cdot x$.

We say that every $x \in |\mathbf{X}|$ *has* finite support.

Henceforth $\mathbf{X}$ and $\mathbf{Y}$ range over nominal renaming sets. x and x' range over elements of $|\mathbf{X}|$ and y and y' range over elements of $|\mathbf{Y}|$ unless stated otherwise.

Definition 5. *Nominal renaming sets form a category* Ren. *Objects are nominal renaming sets. An arrow* $F : \mathbf{X} \longrightarrow \mathbf{Y}$ *is a function* $F \in |\mathbf{X}| \to |\mathbf{Y}|$ *such that* $\sigma \cdot F(x) = F(\sigma \cdot x)$ *always.* F, G, H *will range over arrows.*

For example: $\mathbf{A}$ with action $\sigma \cdot a = \sigma(a)$ is a nominal renaming set. We have $supp(a) = \{a\}$. We will write this renaming set by $\mathbf{A}$, too. In general, we will distinguish notationally between a renaming set $\mathbf{X}$ and its underlying set $|\mathbf{X}|$.

Call $\mathbf{X}$ *trivial* when $\sigma \cdot x = x$ for all $x \in |\mathbf{X}|$ and $\sigma \in Fin$ (so $supp(x) = \emptyset$ always). Write $\mathbf{B} = (\{\top, \bot\}, \cdot)$ for a two-element trivial nominal renaming set. We conclude with one more useful definition:

Definition 6. *Let* $\mathbf{X} \times \mathbf{Y}$ *have underlying set* $|\mathbf{X}| \times |\mathbf{Y}|$ *(that is,* $z \in |\mathbf{X} \times \mathbf{Y}|$ *is a pair* (x, y) *where* $x \in \mathbf{X}$ *and* $y \in \mathbf{Y}$*) and pointwise renaming action (that is,* $\sigma \cdot (x, y) = (\sigma \cdot x, \sigma \cdot y)$*). Call this the* product *of* $\mathbf{X}$ *and* $\mathbf{Y}$.

2.1 Support of Nominal Renaming Sets

Definition 7. *Say* $S \subseteq \mathbf{A}$ supports x *when for all* σ, σ'*, if* $\sigma|_S = \sigma'|_S$ *then* $\sigma \cdot x = \sigma' \cdot x$.

Lemma 8 and Theorem 9 echo [13, Proposition 3.4]. The proofs for nominal renaming set are simpler:

Lemma 8. *If* $S, S' \subseteq \mathbf{A}$ *support* x *then so does* $S \cap S'$.

Proof. Suppose $\sigma|_{S \cap S'} = \sigma'|_{S \cap S'}$. Define $\sigma'' \in Fin$ by: $\sigma''(a) = \sigma(a)$ if $a \in S$, and $\sigma''(a) = \sigma'(a)$ if $a \in \mathbf{A} \setminus S$. $\sigma''|_S = \sigma|_S$ so $\sigma'' \cdot x = \sigma \cdot x$. It is not hard to verify that $\sigma''|_{S'} = \sigma'|_{S'}$ so $\sigma'' \cdot x = \sigma' \cdot x$. The result follows.

Theorem 9. – x *has a unique finite least supporting set* $supp(x)$*; the* support *of* x.

– $\sigma|_{supp(x)} = \sigma'|_{supp(x)}$ *implies* $\sigma \cdot x = \sigma' \cdot x$.

Proof. There is a finite $S \subseteq \mathbf{A}$ supporting x. The first part follows by Lemma 8. The second part is then by Definition 7.

If the reader thinks of fv (free variables of) when they see $supp$, they will not go far wrong [13, Example 6.11]. However, support is an abstract notion valid for any element of any nominal renaming set.

As is standard in nominal techniques we write $a\#x$ for $a \notin supp(x)$, and read it as 'a is fresh for x'. We may write $a\#x, y$ for '$a\#x$ and $a\#y$', and so on.

Definition 10. *Let PFin have underlying set the collection of finite sets of atoms, with pointwise renaming action.*

That is, if $S \subseteq \mathbf{A}$ is finite then $\sigma \cdot S = \{\sigma(a) \mid a \in S\}$. It is not hard to prove that $supp(S) = S$ always.

Lemma 11. *1.* $supp(\sigma \cdot x) \subseteq \sigma \cdot supp(x)$.
2. If σ *is injective on* $supp(x)$ *then* $supp(\sigma \cdot x) = \sigma \cdot supp(x)$.

Proof. For the first part, we will show that $\sigma \cdot supp(x)$ supports $\sigma \cdot x$. The claim then follows. To see the former suppose that $\sigma_1|_{\sigma \cdot supp(x)} = \sigma_2|_{\sigma \cdot supp(x)}$. We then have $(\sigma_1 \circ \sigma)|_{supp(x)} = (\sigma_2 \circ \sigma)|_{supp(x)}$, hence $\sigma_1 \cdot \sigma \cdot x = (\sigma_1 \circ \sigma) \cdot x = (\sigma_2 \circ \sigma) \cdot x = \sigma_2 \cdot \sigma \cdot x$.

For the second part, it suffices to prove the reverse inclusion. Suppose $\sigma|_{supp(x)} = [a_1 \mapsto y_1, \ldots, a_n \mapsto y_n]|_{supp(x)}$. By assumption if $y_i = y_j$ then $i = j$ for $1 \leq i, j \leq n$. So we can form $\sigma' = [y_1 \mapsto a_1, \ldots, y_n \mapsto a_n]$. By Theorem 9 $\sigma' \cdot \sigma \cdot x = x$. By part 1 of this result $supp(\sigma' \cdot \sigma \cdot x) \subseteq \sigma' \cdot supp(\sigma \cdot x)$. Thus, $\sigma \cdot supp(x) = \sigma \cdot supp(\sigma' \cdot \sigma \cdot x) \subseteq \sigma \cdot \sigma' \cdot supp(\sigma \cdot x)$.

Now, if $a \in supp(\sigma \cdot x)$ then by part 1 of this result we can write $a = \sigma(b)$ for some $b \in supp(x)$. So, $\sigma \cdot \sigma' \cdot a = \sigma \cdot \sigma' \cdot \sigma \cdot b = \sigma \cdot b = a$. We have thus shown $\sigma \cdot \sigma' \cdot supp(\sigma \cdot x) \subseteq supp(\sigma \cdot x)$ and hence the claim.

We remark that the extra assumption of injectivity for the second part is not redundant.

Corollary 12 helps to calculate support:

Corollary 12. *If* $a \in supp(x)$ *then* $[a \mapsto b] \cdot x \neq x$. *Taking the contrapositive, if* $[a \mapsto b] \cdot x = x$ *then* $a\#x$. *Note that* $a \neq b$ *is an implicit assumption.*

Proof. By part 1 of Lemma 11 $a \notin supp([a \mapsto b] \cdot x)$. We assumed $a \in supp(x)$, the result follows.

Lemma 13. $S \subseteq \mathbb{A}$ *supports* x *if and only if* $\sigma|_S = id|_S$ *implies* $\sigma \cdot x = x$ *for all* σ.

Proof. The left-to-right implication is direct from the definition of 'supports'. For the right-to-left implication we will show $supp(x) \subseteq S$. To that end, pick $a \in supp(x)$ and $b \notin supp(x)$. From Corollary 12 we then obtain $\sigma \cdot x \neq x$ for $\sigma = [a \mapsto b]$. Hence $\sigma|_S \neq id|_S$, hence $a \in S$.

We need Lemma 14 for Lemma 33 and Corollary 29:

Lemma 14. *Suppose* $c\#x$ *and* $c\#x'$.
Then $[a \mapsto c] \cdot x = [a' \mapsto c] \cdot x'$ *if and only if* $a'\#x$ *and* $x' = [a \mapsto a'] \cdot x$.

Proof. Suppose $a'\#x$ and $x' = [a{\mapsto}a'] \cdot x$. We reason as follows:

$$\begin{array}{lll} [a'{\mapsto}c] \cdot x' & = [a'{\mapsto}c] \cdot [a{\mapsto}a'] \cdot x & x' = [a{\mapsto}a'] \cdot x \\ & = [a{\mapsto}c] \cdot x & \text{Theorem 9, } a'\#x \end{array}$$

Conversely if $[a{\mapsto}c] \cdot x = [a'{\mapsto}c] \cdot x'$ then $[c{\mapsto}a'] \cdot [a{\mapsto}c] \cdot x = [c{\mapsto}a'] \cdot [a'{\mapsto}c] \cdot x'$. By Theorem 9 $[c{\mapsto}a'] \cdot [a'{\mapsto}c] \cdot x' = x'$ and $[c{\mapsto}a'] \cdot [a{\mapsto}c] \cdot x = [a{\mapsto}a'] \cdot x$. It follows that $x' = [a{\mapsto}a'] \cdot x$. By similar reasoning $x = [a'{\mapsto}a] \cdot x'$, and by part 1 of Lemma 11 it follows that $a'\#x$.

3 The Exponential

Definition 15. *Let* $|\mathbf{X} \Rightarrow \mathbf{Y}|$ *be the set of functions* $f \in |\mathbf{X}| \to |\mathbf{Y}|$ *such that there exists some finite* $S_f \subseteq \mathbf{A}$ *(for each* f*, we fix one such* S_f*) such that for all* $\sigma \in Fin$ *and* $x \in |\mathbf{X}|$ *if* $\sigma|_{S_f} = id|_{S_f}$ *then*

$$\sigma \cdot f(x) = f(\sigma \cdot x). \tag{4}$$

$|\mathbf{X} \Rightarrow \mathbf{Y}|$ serves as an underlying set in Definition 20. First, we consider some examples and prove properties of $|\mathbf{X} \Rightarrow \mathbf{Y}|$.

Remark 16. $-\,\pi_1 \in |\mathbf{X} \times \mathbf{Y}| \to |\mathbf{X}|$ mapping (x, y) to x is in $|(\mathbf{X} \times \mathbf{Y}) \Rightarrow \mathbf{X}|$.
$-$ The map $= \in |\mathbf{A} \times \mathbf{A}| \to |\mathbf{B}|$ mapping (x, y) to $\top$ if $x = y$ and to $\bot$ if $x \neq y$, is *not* an element of $|(\mathbf{A} \times \mathbf{A}) \Rightarrow \mathbf{B}|$. Unpacking definitions, the reason is that there is no finite $S \subseteq \mathbf{A}$ such that if $\sigma|_S = id|_S$ then for all $x, y \in \mathbf{A}$, $x = y$ if and only if $\sigma(x) = \sigma(y)$.
$-$ $supp_{\mathbf{X}} \in |\mathbf{X}| \to |PFin|$ mapping x to $supp(x)$, may or may not be an element of $|\mathbf{X} \Rightarrow PFin|$. If $\mathbf{X} = \mathbf{A}$ then $supp(a) = \{a\}$ and $\sigma \cdot supp(a) = \sigma \cdot \{a\} = \{\sigma(a)\} = supp(\sigma(a))$. Therefore $supp_{\mathbf{A}} \in |\mathbf{A} \Rightarrow PFin|$. Similarly for $supp_{PFin}$.
On the other hand if we let $\mathbf{X}$ have $|\mathbf{X}| = |PFin|$ (finite sets of atoms) and the renaming action such that $\sigma \cdot S = \{\sigma(a) \mid a \in S\}$ if $\sigma|_S$ is injective, and $\sigma \cdot S = \emptyset$ otherwise, then $supp_{\mathbf{X}} \notin |\mathbf{X} \Rightarrow PFin|$.

Remark 17. Intuitively, a map that does not compare atoms in its argument for inequality will be in the underlying set of the exponential. A map that compares atoms for inequality, will not. See the Conclusions for further discussion.

Elements of $|\mathbf{X} \Rightarrow \mathbf{Y}|$ are determined by their 'asymptotic behaviour':

Lemma 18. *Suppose* $f \in |\mathbf{X} \Rightarrow \mathbf{Y}|$ *and* $g \in |\mathbf{X} \Rightarrow \mathbf{Y}|$*. Suppose also* $S \subseteq \mathbf{A}$ *is finite and assume that for all* $x \in |\mathbf{X}|$ *if* $supp(x) \cap S = \emptyset$ *then* $f(x) = g(x)$*. Then* $f = g$*.*

Proof. Choose any $x \in |\mathbf{X}|$. There are two cases:
$-$ If $supp(x) \cap S = \emptyset$ then by assumption $f(x) = g(x)$.

– If $supp(x) \cap S \neq \emptyset$ then let $C = \{c_1, \ldots, c_k\}$ be a fresh choice of atoms (so $C \cap (S \cup S_f \cup S_g \cup supp(x)) = \emptyset$). Let $\tau = [a_1 \mapsto c_1, \ldots, a_k \mapsto c_k]$ and $\tau^{\mathsf{T}} = [c_1 \mapsto a_1, \ldots, c_k \mapsto a_k]$. By Lemma 11 $supp(\tau \cdot x) \cap S = \emptyset$. We reason as follows:

$$\begin{aligned} f(x) &= f(\tau^{\mathsf{T}} \cdot \tau \cdot x) && \text{Theorem 9} \\ &= \tau^{\mathsf{T}} \cdot f(\tau \cdot x) && (4), \text{Definition 15} \\ &= \tau^{\mathsf{T}} \cdot g(\tau \cdot x) && \text{Assumption} \\ &= g(\tau^{\mathsf{T}} \cdot \tau \cdot x) && (4), \text{Definition 15} \\ &= g(x) && \text{Theorem 9} \end{aligned}$$

An element f of the function space can be reconstructed from 'asymptotic' behaviour:

Lemma 19. *Suppose f is a partial function from $|\mathbf{X}|$ to $|\mathbf{Y}|$. Suppose $S \subseteq \mathbf{A}$ is finite and: $supp(x) \cap S = \emptyset$ implies $f(x)$ is defined, and $\sigma|_S = id|_S$ implies $\sigma \cdot f(x) = f(\sigma \cdot x)$, where both are defined.*

Then there is a unique $f' \in |\mathbf{X} \Rightarrow \mathbf{Y}|$ extending f (so $f'(x) = f(x)$ if $f(x)$ is defined).

Proof. First, we define f'. Consider $x \in |\mathbf{X}|$, write $supp(x) = \{a_1, \ldots, a_k\} = S$. Let $C = \{c_1, \ldots, c_k\}$ be fresh atoms (so $C \cap supp(x) = \emptyset = C \cap S$). Define

$$\tau = [a_1 \mapsto c_1, \ldots, a_k \mapsto c_k] \quad \tau^{\mathsf{T}} = [c_1 \mapsto a_1, \ldots, c_k \mapsto a_k] \quad \text{and} \quad f'(x) = \tau^{\mathsf{T}} \cdot f(\tau \cdot x).$$

We first show the choice of fresh C does not matter. Suppose $C' = \{c'_1, \ldots, c'_k\}$ is also fresh (so $C' \cap supp(x) = \emptyset = C' \cap S$). We put $\tau' = [a_1 \mapsto c'_1, \ldots, a_k \mapsto c'_k]$ and $\tau'^{\mathsf{T}} = [c'_1 \mapsto a_1, \ldots, c'_k \mapsto a_k]$. We must show $\tau^{\mathsf{T}} \cdot f(\tau \cdot x) = \tau'^{\mathsf{T}} \cdot f(\tau' \cdot x)$. We assume the special case that

$$C \cap C' = \emptyset \quad \text{and} \quad C' \cap supp(f(\tau \cdot x)) = \emptyset. \tag{5}$$

The general case follows by two applications of the special case for an 'even fresher' set of fresh atoms C''. We write $\mu = [c_1 \mapsto c'_1, \ldots, c_k \mapsto c'_k]$ and $\mu^{\mathsf{T}} = [c'_1 \mapsto c_1, \ldots, c'_k \mapsto c_k]$. By Theorem 9 and $C \cap supp(x) = \emptyset$ we have $\tau' \cdot x = \mu \cdot \tau \cdot x$. Also, by $C' \cap S = \emptyset$ we have $f(\tau' \cdot x) = \mu \cdot f(\tau \cdot x)$. Therefore $\tau'^{\mathsf{T}} \cdot f(\tau' \cdot x) = \tau^{\mathsf{T}} \cdot \mu^{\mathsf{T}} \cdot \mu \cdot f(\tau \cdot x)$.

We recall (5); by Theorem 9 and $C' \cap supp(f(\tau \cdot x)) = \emptyset$, $\mu^{\mathsf{T}} \cdot \mu \cdot f(\tau \cdot x) = f(\tau \cdot x)$. The claim that the choice of fresh C does not matter, follows.

To see that f' indeed extends f we suppose that $x \in |\mathbf{X}|$ and that $f(x)$ is defined. Then $supp(\tau \cdot x) \cap S = \emptyset$ so $f(\tau \cdot x)$ is also defined. By assumption on f we have $\tau^{\mathsf{T}} \cdot f(\tau \cdot x) = f(\tau^{\mathsf{T}} \cdot \tau \cdot x) = f(x)$ where the last equality uses Theorem 9.

To see that $f' \in |\mathbf{X} \Rightarrow \mathbf{Y}|$ we put $S_{f'} = S$. Suppose that $\sigma|_S = id|_S$. We have seen that the choice of the fresh C does not matter so that we can assume that τ and τ^{T} commute with σ. We then have $\sigma \cdot f'(x) = \sigma \cdot \tau^{\mathsf{T}} \cdot f(\tau \cdot x) = \tau^{\mathsf{T}} \cdot \sigma \cdot f(\tau \cdot x) = f'(\sigma \cdot x)$.

Uniqueness of f' is now by Lemma 18.

Definition 20. *Let $\mathbf{X} \Rightarrow \mathbf{Y}$ have underlying set $|\mathbf{X} \Rightarrow \mathbf{Y}|$ with renaming action*

$$\text{if} \quad \sigma \cdot x = x \quad \text{then} \quad (\sigma \cdot f)x = \sigma \cdot f(x). \tag{6}$$

By Lemma 19 this uniquely determines the renaming action for all $x \in |\mathbf{X}|$.

Lemma 21. $\mathbf{X} \Rightarrow \mathbf{Y}$ *is a nominal renaming set.*

Proof. $f \in |\mathbf{X} \Rightarrow \mathbf{Y}|$ is supported by the set S_f from Definition 15, for suppose that $\sigma|_{S_f} = id|_{S_f}$. By Lemma 13 it suffices to show $\sigma \cdot f = f$. Now put $S = S_f \cup \{b \mid \sigma(b) \neq b\}$. This is a finite set so by Lemma 18 it suffices to show $(\sigma \cdot f)(x) = f(x)$ for all x such that $supp(x) \cap S = \emptyset$. Fix such an x. By Theorem 9 $\sigma \cdot x = x$ so by (6) we know $(\sigma \cdot f)(x) = \sigma \cdot f(x)$. $supp(x) \cap S_f = \emptyset$ by assumption so by (4) we know $\sigma \cdot f(x) = f(\sigma \cdot x)$. Then $f(\sigma \cdot x) = f(x)$ since $\sigma \cdot x = x$. The result follows.

Theorem 22. $\sigma \cdot f(x) = (\sigma \cdot f)(\sigma \cdot x)$, *for* $f \in |\mathbf{X} \Rightarrow \mathbf{Y}|$.

Proof. Write $supp(f) \cup supp(\sigma \cdot f) \cup \{a \mid \sigma(a) \neq a\} = \{a_1, \ldots, a_k\}$. Choose fresh $C = \{c_1, \ldots, c_k\}$, so $C \cap (supp(x) \cup S_f \cup S_{\sigma \cdot f}) = \emptyset$ and $\sigma|_C = id|_C$. Define

$$\tau = [a_1 \mapsto c_1, \ldots, a_k \mapsto c_k] \qquad \tau^{\mathsf{T}} = [c_1 \mapsto a_1, \ldots, c_k \mapsto a_k]$$
$$\pi = [a_1 \mapsto c_1, c_1 \mapsto a_1, \ldots, a_k \mapsto c_k, c_k \mapsto a_k] \qquad \sigma' = \pi \circ \sigma \circ \pi.$$

Then:

$$(\tau^{\mathsf{T}} \circ \sigma' \circ \tau)|_{supp(x)} = \sigma|_{supp(x)} \quad C \cap supp(x) = \emptyset \tag{7}$$
$$\tau^{\mathsf{T}} \circ \sigma' \circ \sigma = \sigma \circ \tau^{\mathsf{T}} \quad \sigma(c){=}c \text{ for all } c{\in}C \tag{8}$$
$$(\tau^{\mathsf{T}} \circ \sigma')|_{S_{\sigma \cdot f}} = id|_{S_{\sigma \cdot f}} \quad \text{By construction} \tag{9}$$

So:

$$\begin{aligned}
(\sigma \cdot f)(\sigma \cdot x) &= (\sigma \cdot f)((\tau^{\mathsf{T}} \circ \sigma') \cdot \tau \cdot x) && \text{Th. 9 and (7)} \\
&= (\tau^{\mathsf{T}} \circ \sigma') \cdot (\sigma \cdot f)(\tau \cdot x) && \text{(9) and (4), Def. 15} \\
&= (\tau^{\mathsf{T}} \circ \sigma') \cdot \sigma \cdot f(\tau \cdot x) && \text{(9) and (6), Def. 20} \\
&= (\sigma \circ \tau^{\mathsf{T}}) \cdot f(\tau \cdot x) && \text{(8)} \\
&= \sigma \cdot f(\tau^{\mathsf{T}} \cdot \tau \cdot x) && \text{(4), Def. 15} \\
&= \sigma \cdot f(x) && \text{Theorem 9}
\end{aligned}$$

Corollary 23. *If* $f{\in}|\mathbf{X}{\Rightarrow}\mathbf{Y}|$ *then the following are equivalent:*

– *For all* $x \in |\mathbf{X}|$ *and* $\sigma \in Fin$, *if* $\sigma|_S = id|_S$ *then* $\sigma{\cdot}f(x) = f(\sigma{\cdot}x)$.
– $supp(f) \subseteq S$.

Proof. If $\sigma|_{supp(f)} = id|_{supp(f)}$ then by Theorems 22 and 9, $\sigma \cdot f(x) = (\sigma \cdot f)(\sigma \cdot x) = f(\sigma \cdot x)$.

Now suppose $S \subseteq \mathbf{A}$ and for all $x \in |\mathbf{X}|$ if $\sigma|_S = id|_S$ then $\sigma{\cdot}f(x) = f(\sigma{\cdot}x)$. If we show that S supports f then by the 'unique least' property of support (Theorem 9) we are done. Suppose $\sigma|_S = id|_S$ and take any $x \in |\mathbf{X}|$ such that $\sigma|_{supp(x)} = id|_{supp(x)}$:

$$f(x) \overset{\text{Theorem 9}}{=} f(\sigma \cdot x) \overset{\text{Assumption}}{=} \sigma \cdot f(x) \overset{\text{Theorem 22}}{=} (\sigma \cdot f)(\sigma \cdot x) \overset{\text{Theorem 9}}{=} (\sigma \cdot f)(x).$$

By Lemma 18 $\sigma \cdot f = f$ as required.

Compare Corollary 24 with [13, Example 4.9, (24)]:

Corollary 24. *$supp(f(x)) \subseteq supp(f) \cup supp(x)$ always, for $f \in |\mathbf{X} \Rightarrow \mathbf{Y}|$.*

Proof. Using Theorems 22 and 9.

Lemma 25. *Arrows $F : \mathbf{X} \longrightarrow \mathbf{Y}$ are exactly $f \in |\mathbf{X} \Rightarrow \mathbf{Y}|$ such that $supp(f) = \emptyset$.*

Proof. $F : \mathbf{X} \longrightarrow \mathbf{Y}$ is a map in $|\mathbf{X}| \to |\mathbf{Y}|$. By definition $\sigma \cdot F(x) = F(\sigma \cdot x)$. By Corollary 23, $supp(F) = \emptyset$. Conversely if $f \in |\mathbf{X} \Rightarrow \mathbf{Y}|$ and $supp(f) = \emptyset$ then f is a map in $|\mathbf{X}| \to |\mathbf{Y}|$. Also, $\sigma \cdot f(x) = f(\sigma \cdot x)$ by Theorems 22 and 9.

Theorem 26. - $\Rightarrow$ - is *an exponential in* Ren.

Proof. We show that currying and uncurrying are arrows between $(\mathbf{X} \times \mathbf{Y}) \longrightarrow \mathbf{Z}$ and $\mathbf{X} \longrightarrow (\mathbf{Y} \Rightarrow \mathbf{Z})$. The $\beta\eta$ equations are then inherited.
Currying: Take $F \in (\mathbf{X} \times \mathbf{Y}) \longrightarrow \mathbf{Z}$. For $x \in \mathbf{X}$ we put $F_x = (\lambda y{\in}|\mathbf{Y}|.F(x,y))$. We show that $\lambda x{\in}|\mathbf{X}|.F_x \in \mathbf{X} \longrightarrow (\mathbf{Y} \Rightarrow \mathbf{Z})$:

If $x \in |X|$ we put $S_{F_x} = supp(x)$. If $\sigma|_{supp(x)} = id|_{supp(x)}$ then $\sigma \cdot F_x(y) = F(\sigma \cdot x, \sigma \cdot y) = F(x, \sigma \cdot y) = F_x(\sigma \cdot y)$, so $F_x \in |\mathbf{X} \Rightarrow \mathbf{Y}|$.

To see that $x \mapsto F_x$ is an arrow pick arbitrary σ. We need to show $\sigma \cdot F_x = F_{\sigma \cdot x}$. Appealing to Lemma 18 we choose $y \in \mathbf{Y}$ with $\sigma \cdot y = y$ and $supp(y)$ disjoint from $supp(x) = supp(F_x)$. We then have $(\sigma \cdot F_x)(y) = \sigma \cdot F(x, \sigma \cdot y) = \sigma \cdot F(x,y) = F(\sigma \cdot x, \sigma \cdot y) = F(\sigma \cdot x, y) = F_{\sigma \cdot x}(y)$.
Uncurrying: Take $G \in \mathbf{X} \longrightarrow (\mathbf{Y} \Rightarrow \mathbf{Z})$. We show that $\lambda(x,y){\in}|\mathbf{X} \times \mathbf{Y}|.G(x)(y) \in (\mathbf{X} \times \mathbf{Y}) \longrightarrow \mathbf{Z}$:

It suffices to show that $\sigma \cdot G(x)(y) = G(\sigma \cdot x)(\sigma \cdot y)$. Now $\sigma \cdot G(x)(y) = (\sigma \cdot G(x))(\sigma \cdot y)$ by Theorem 22 and $\sigma \cdot G(x) = G(\sigma \cdot x)$ by Definition 5.

Lemma 27 does for Ren what [13, Lemma 6.3] does for the category of nominal sets. We can develop similar inductive and recursive principles for syntax-with-binding as are exhibited using the Gabbay-Pitts И-quantifier in Theorem 6.5 in [13].

Lemma 27. *There is a bijection between $F : (\mathbf{A} \times \mathbf{X}) \longrightarrow \mathbf{Y}$ such that $a\#F(a,x)$ always, and $G \in (\mathbf{A} \Rightarrow \mathbf{X}) \longrightarrow \mathbf{Y}$.*

Proof. We define mappings as follows:

– F maps to $\lambda f \in |\mathbf{A} \Rightarrow \mathbf{X}|.Иa.\ F(a, fa)$ where (as is standard [13]) $Иa.\ F(a,fa)$ is equal to $F(a,fa)$ for any a such that $a\#f$.
– G maps to $\lambda(a,x) \in |\mathbf{A} \times \mathbf{X}|.G(\lambda y{\in}\mathbf{A}.[a{\mapsto}y] \cdot x)$.

We can prove this is well-defined, and the result follows by calculations.

4 The Atoms-Exponential $\mathbf{A} \Rightarrow \mathbf{X}$

Lemma 28. *If $f \in |\mathbf{A} \Rightarrow \mathbf{X}|$ then $f = \lambda y{\in}\mathbf{A}.[a{\mapsto}y] \cdot x$ for some $x \in |\mathbf{X}|$ and $a\#f$.*

Proof. By Definition 20 $f \in |\mathbf{A} \Rightarrow \mathbf{X}|$ when $f \in |\mathbf{A}| \to |\mathbf{X}|$ and there is a finite $S \subseteq \mathbf{A}$ such that $\sigma \in Fin$, $a \in \mathbf{A}$, and $\sigma|_S = id|_S$ imply $\sigma \cdot f(a) = f(\sigma(a))$. Choose $a \notin S$ and $y \in \mathbf{A}$. Then $f(y) = f([a{\mapsto}y] \cdot a) = [a{\mapsto}y] \cdot f(a)$, and we take $x = f(a)$.

Two functions on atoms are equal if they agree for one fresh atom; compare this for example with axiom (Ext^{τ}_{ν}) in the theory of contexts [17, Figure 4] and the extensionality principle for concretion of the Gabbay-Pitts model of atoms-abstraction in nominal sets [13, Proposition 5.5, equation (48)]:

Corollary 29. *Suppose $f, f' \in |\mathbf{A} \Rightarrow \mathbf{X}|$ and suppose $c\#f$ and $c\#f'$. Then $f(c) = f'(c)$ if and only if $f = f'$.*

Proof. The right-to-left implication is easy. Assume $f(c) = f'(c)$. By Lemma 28
- there exist $x \in |\mathbf{X}|$ and $a \in \mathbf{A}$ such that $a\#f$ and $f = \lambda y{\in}\mathbf{A}.[a{\mapsto}y] \cdot x$, and
- there exist $x' \in |\mathbf{X}|$ and $a' \in \mathbf{A}$ such that $a'\#f'$ and $f' = \lambda y{\in}\mathbf{A}.[a'{\mapsto}y] \cdot x'$.

We assumed that $f(c) = f'(c)$ so $[a{\mapsto}c] \cdot x = [a'{\mapsto}c] \cdot x'$. By Lemma 14 $a'\#x$ and $x' = [a{\mapsto}a'] \cdot x$. Choose any $y \in \mathbf{A}$. We reason as follows:

$$\begin{array}{ll} f'(y) = [a'{\mapsto}y] \cdot x' & f' = \lambda y{\in}\mathbf{A}.[a'{\mapsto}y] \cdot x' \\ = [a'{\mapsto}y] \cdot [a{\mapsto}a'] \cdot x. & x' = [a{\mapsto}a'] \cdot x \\ = [a{\mapsto}y] \cdot x & \text{Theorem 9, } a'\#x \end{array}$$

Lemma 30. *Suppose $f \in |\mathbf{A} \Rightarrow \mathbf{Y}|$. Then $a\#f$ if and only if $a\#f(b)$, for any $b\#f$ (by our permutative convention, $b \neq a$).*

Equivalently, $supp(f) = supp(f(b)) \setminus \{b\}$ for any $b\#f$.

Proof. We prove two implications. Choose any $b\#f$.

If $a\#f$ then by Corollary 24 $a\#fb$ and we are done. Suppose that $a\#f(b)$. We have assumed $b\#f$ so by Corollary 12 it suffices to prove $[a{\mapsto}b]\cdot f = f$. Choose a fresh c (so $c\#f$ and $c\#[a{\mapsto}b] \cdot f$). By Corollary 29 it suffices to check that $f(c) = ([a{\mapsto}b] \cdot f)(c)$. Note that by Corollary 24 $a\#f(c)$. We reason as follows:

$$\begin{array}{ll} fc = [a{\mapsto}b] \cdot f(c) & \text{Theorem 9 and Corollary 24} \\ = ([a{\mapsto}b] \cdot f)([a{\mapsto}b] \cdot c) & \text{Theorem 22} \\ = ([a{\mapsto}b] \cdot f)c & [a{\mapsto}b] \cdot c = c \end{array}$$

Lemma 31. *$\lambda y{\in}\mathbf{A}.[a{\mapsto}y]\cdot x$ is supported by $supp(x)$; thus $\lambda y{\in}\mathbf{A}.[a{\mapsto}y]\cdot x{\in}|\mathbf{A}{\Rightarrow}\mathbf{X}|$.*

Proof. The corollary is immediate given the first part, by Definition 15. We now prove that $supp(x)$ supports $\lambda y{\in}\mathbf{A}.[a{\mapsto}y]\cdot x$. By Corollary 23 it suffices to show $\sigma|_{supp(x)} = id|_{supp(x)}$ implies $\sigma \cdot \lambda y{\in}\mathbf{A}.[a{\mapsto}y] \cdot x = \lambda y{\in}\mathbf{A}.[a{\mapsto}y] \cdot x$.

So suppose $\sigma|_{supp(x)} = id|_{supp(x)}$. By Corollary 29 it suffices to check

$$(\sigma \cdot \lambda y{\in}\mathbf{A}.[a{\mapsto}y] \cdot x)c = (\lambda y{\in}\mathbf{A}.[a{\mapsto}y] \cdot x)c$$

for fresh c. Choose c such that $c\#\sigma \cdot \lambda y{\in}\mathbf{A}.[a{\mapsto}y] \cdot x$, $c\#\lambda y{\in}\mathbf{A}.[a{\mapsto}y] \cdot x$, $\sigma(c) = c$ and $c\#x$. By Theorem 22 since $\sigma(c) = c$ we have that

$$\sigma \cdot ((\lambda y{\in}\mathbf{A}.[a{\mapsto}y] \cdot x)c) = (\sigma \cdot \lambda y{\in}\mathbf{A}.[a{\mapsto}y] \cdot x)c.$$

So it suffices to check that $\sigma\cdot[a{\mapsto}c]\cdot x = [a{\mapsto}c]\cdot x$. We assumed $\sigma|_{supp(x)} = id|_{supp(x)}$ and $c\#x$. It follows that $(\sigma \circ [a{\mapsto}c])|_{supp(x)} = [a{\mapsto}c]|_{supp(x)}$. By Theorem 9 the result follows.

Corollary 32. $supp(\lambda y{\in}\mathbf{A}.[a{\mapsto}y]\cdot x) = supp(x)\setminus\{a\}$.

Proof. Choose some fresh b (so $b\#\lambda y{\in}\mathbf{A}.[a{\mapsto}y]\cdot x$ and $b\#x$). By Lemma 31 $\lambda y{\in}\mathbf{A}.[a{\mapsto}y]\cdot x \in |\mathbf{A}\Rightarrow\mathbf{X}|$. Therefore by Lemma 30 we know that

$$supp(\lambda y{\in}\mathbf{A}.[a{\mapsto}y]\cdot x) = supp([a{\mapsto}b]\cdot x)\setminus\{b\}.$$

Since $b\#x$ by part 2 of Lemma 11 $supp([a{\mapsto}b]\cdot x) = (supp(x)\setminus\{a\})\cup\{b\}$.

Lemma 33. *Suppose* $x, x' \in |\mathbf{X}|$, *and* $a, a' \in \mathbf{A}$. *Then* $\lambda y{\in}\mathbf{A}.[a{\mapsto}y]\cdot x = \lambda y{\in}\mathbf{A}.[a'{\mapsto}y]\cdot x'$ *if and only if* $a'\#x$ *and* $x' = [a{\mapsto}a']\cdot x$.

Proof. Using Lemma 14 and Corollary 29.

The reader may recognise these results from the theory of the Gabbay-Pitts model of atoms-abstraction in nominal sets [13, Definition 5.4]. A nominal renaming set is also a nominal permutation set (by 'forgetting' the action for non-bijective σ); using Corollary 12 it can be proved [11, Theorem 4.8] that the notions of support coincide. In the spirit of [13, (35)] we can write[3] $[a]x = \{(a,x)\}\cup\{(c,[a{\mapsto}c]\cdot x) \mid c\#x\}$ and $|[\mathbf{A}]\mathbf{X}| = \{[a]x \mid a\in\mathbf{A},\ x\in|\mathbf{X}|\}$. The proof of Theorem 34 is now routine [11, Theorem 5.7]:

Theorem 34. $|\mathbf{A}\Rightarrow\mathbf{X}|$ *is in bijective correspondence with* $|[\mathbf{A}]\mathbf{X}|$. *Inverse mappings are given by:*

- α *maps* $z\in|[\mathbf{A}]\mathbf{X}|$ *to* $\lambda y{\in}\mathbf{A}.[a{\mapsto}y]\cdot x$, *for* $(a,x){\in}z$.
- β *maps* $f\in|\mathbf{A}\Rightarrow\mathbf{X}|$ *to* $[a](fa)$, *for* $a\#f$.

5 Presheaf and Topos Structure of Ren

For convenience, take the finite sets in $\mathbb{I}$ and $\mathbb{F}$ from the Introduction to be finite $S\subseteq\mathbb{A}$.

We could now go on and exhibit the subobject classifier in Ren, thus establishing that Ren is a topos, and hence is a model of higher-order logic. This is not hard to do: a subobject classifier is the renaming set Ω with underlying set $|\Omega|$ those $U\subseteq Fin$ such that:

- If $\sigma\in U$ then $\mu\circ\sigma\in U$ for all $\mu\in Fin$.
- There exists finite $S\subseteq\mathbf{A}$ such that $\mu\in U$ and $\sigma|_S = id|_S$ imply $\mu\circ\sigma\in U$.

The renaming action is given by $\sigma\cdot U = \{\mu \mid \mu\circ\sigma\in U\}$. The proof that this makes Ren into a topos is by standard calculations. We also remark that Ren is a Grothendieck topos for a topology on the opposite of the category of finite sets and functions and thus a full subcategory of the functor category $\mathsf{Set}^{\mathbb{F}}$. The required topology has for basic covers of an object X nonempty families of monos $f_i : X \longrightarrow X_i$. The interested reader is referred to [24] where a proof of a similar result is worked out.

We now characterise Ren as a category of presheaves.

[3] The use of $[a{\mapsto}c]$ instead of the swapping $(a\ c)$ (swapping defined in [13, (3)]) is immaterial because $c\#x$ and so the actions coincide by Theorem 9.

Definition 35. *Let* PBM *be the category of presheaves in* $\mathsf{Set}^{\mathbb{F}}$ *that preserve pullbacks of pairs of monos, and natural transformations between them.*

If $f \in X \to Y$ is a function let its *image* $img(f)$ be $\{f(x) \mid x \in X\} \subseteq Y$. Note that monos in $\mathbb{F}$ and Ren are injections, and that a pullback of a pair of monos is given by set intersection with the natural inclusion maps. Write '$\iota : S \subseteq S'$' for "$S \subseteq S'$ and we write ι for the subset inclusion arrow in $\mathbb{F}$".

Definition 36. *Fix* $F \in$ PBM. *Let* (S, x) *range over pairs where* $S \subseteq \mathbf{A}$ *is finite and* $x \in F(S)$. *Write* $\sim$ *for the least equivalence relation such that* $\iota : S \subseteq S'$ *implies* $(S, x) \sim (S', F(\iota)(x))$.

Lemma 37. *If* $\iota_1 : S_1 \subseteq S'$ *and* $\iota_2 : S_2 \subseteq S'$ *and* $(S_1, x_1) \sim (S', x') \sim (S_2, x_2)$ *then there exists some* $x \in F(S_1 \cap S_2)$ *such that* $(S_1 \cap S_2, x) \sim (S', x')$. *Thus, each* $\sim$*-equivalence class has a unique least representative* (S, x), *'least' in the sense that if* $(S, x) \sim (S', x')$ *then* $\iota : S \subseteq S'$ *and* $x' = F(\iota)(x)$.

Theorem 38. Ren *is equivalent to* PBM.

Proof. $\mathbf{X}$ maps to $F_{\mathbf{X}}$ mapping $S \in \mathbb{F}$ to $\{x \in |\mathbf{X}| \mid supp(x) \subseteq S\}$ and mapping $\tau : S \to S'$ to the renaming action of τ extended to a total function which is the identity off S. This is a presheaf by part 1 of Lemma 11 and we can prove that it preserves pullbacks of monos using Lemma 8. F maps to the set of unique least representative elements of $\sim$-equivalence classes, as constructed in Lemma 37 with action given by $\tau \cdot (S, x)$ is the representative of the $\sim$-equivalence class of $(\tau \cdot S, F(\tau|_S)(x))$. The result follows by routine calculations.

Preserving pullbacks of monos, and the corollary that we have 'least representatives', makes possible the sets-based presentation of nominal sets and nominal renaming sets, contrasting with the purely presheaf-based presentations of [7,16].

6 Tripos Structure on Ren

As argued in [16] the topos logic of $\mathcal{F}$, thus also that of Ren, is unsuited to reasoning about syntax with binding. For example equality of atoms is not decidable in that logic. In fact, in the topos logic of Ren and $\mathcal{F}$, the proposition $\forall x, y{\in}\mathbf{A}.\neg\neg(x{=}y)$ holds.

It was proposed in [16] to use the tripos obtained by pulling back the logic from the Schanuel topos instead. In particular, the tripos so obtained is needed to justify the theory of contexts [17] (well — *almost*; the restriction to pullback-preserving functors had not been made explicit in that paper.)

In keeping with our goal of making these constructions concrete and usable for direct calculations, we make this construction explicit at the level of Ren.

Definition 39. *For a nominal renaming set* $\mathbf{X}$ *let* $Pred(\mathbf{X})$ *be the set of those subsets* U *of* $|\mathbf{X}|$ *that are preserved by bijective renamings, i.e., for which* $x \in U$ *implies* $\sigma \cdot x \in U$ *provided that* $\sigma \in Fin$ *is bijective, i.e., a permutation.*

$Pred(\mathbf{X})$ *is ordered by subset inclusion and* $Pred(-)$ *extends to a functor* $\mathsf{Ren}^{op} \to$ *Poset where* $Pred(f)(U) = \{x \mid f(x) \in U\}$ *when* $f : \mathbf{X} \to \mathbf{Y}$ *and* $U \in Pred(\mathbf{Y})$.

We remark that in the topos logic of Ren a predicate is a subset preserved by all renamings, not just the bijective ones.

Theorem 40 is a consequence of the folklore result stated in [16] which asserts that triposes can be pulled back along finite-limit-preserving functors with a right adjoint ('geometric morphisms'). Rather than detailing this argument we illustrate the tripos structure by concrete constructions below.

Theorem 40. *The pair* $(\mathsf{Ren}, Pred)$ *forms a tripos.*

This means there is enough structure to interpret higher-order logic. In particular, predicates are closed under boolean operations, universal quantification over renaming sets, and there is a renaming set $\mathbf{O}$ of propositions such that morphisms $\mathbf{X} \to \mathbf{O}$ are in 1-1 correspondence with elements of $Pred(\mathbf{X})$. We explicitly construct these ingredients in order to demonstrate that the logic for Ren is quite close to classical set-theoretic reasoning in contrast to the functorial reasoning that was used in [17].

Proposition 41. *The object of propositions* $\mathbf{O}$ *in the tripos* $(\mathsf{Ren}, Pred)$ *has underlying set those* $U \subseteq Fin$ *such that:*

– *If* $\sigma \in U$ *and* $\pi \in Fin$ *is a permutation then* $\pi \circ \sigma \in U$

– *There exists finite* $S \subseteq \mathbf{X}$ *such that if* $\sigma|_S = \sigma'|_S$ *then* $\sigma \in U$ *iff* $\sigma' \in U$.

The renaming action in $\mathbf{O}$ *is given by* $\sigma \cdot U = \{\mu \mid \mu \circ \sigma \in U\}$.

Proof (Sketch). If $P \in Pred(\mathbf{X})$ then a morphism $\chi_P : \mathbf{X} \to \mathbf{O}$ is defined by $\chi_P(a) = \{\sigma \mid \sigma \cdot a \in P\}$. Conversely, if $m : \mathbf{X} \to \mathbf{O}$ is a morphism then we associate the predicate $P_m = \{a \mid m(a) = \top\}$ where $\top \in \mathbf{O}$ is given by $\top = Fin$.

Proposition 42. *The logic of* $(\mathsf{Ren}, Pred)$ *is classical.*

Proof. It suffices to show that if $P \in Pred(\mathbf{X})$ is a predicate then its set-theoretic complement $\{x \mid x \notin P\}$ is again a predicate. But if $\pi \in Fin$ is a permutation and $x \notin P$ then $\pi \cdot x \notin P$ for otherwise $x = \pi^{-1} \cdot \pi \cdot x \in P$.

Proposition 43 (universal quantification). *Let* $P \in Pred(\mathbf{X} \times \mathbf{Y})$ *(possibly given by a morphism* $\mathbf{X} \times \mathbf{Y} \to \mathbf{O}$*). Its universal quantification* $\forall P \in Pred(\mathbf{X})$ *is given by* $\forall P = \{x \mid \forall y{\in}\mathbf{Y}.(x, y) \in P\}$.

Proof. Suppose that $Q \in Pred(\mathbf{X})$. We have to show that $\forall P$ is a predicate and that the following are equivalent:

– for all $x \in \mathbf{X}$ and $y \in \mathbf{Y}$ if $x \in Q$ then $(x, y) \in P$

– for all $x \in \mathbf{X}$ if $x \in Q$ then $x \in \forall P$

The equivalence being obvious from the definition it only remains to show that $\forall P$ is indeed a predicate. So assume that $\pi \in Fin$ is a permutation and consider that $x \in \forall P$. If $y \in \mathbf{Y}$ then $(x, \pi^{-1} \cdot y) \in P$ since $x \in \forall P$. So, $(\pi \cdot x, y) \in P$ since P is a renaming set. Hence, since y was arbitrary, $\pi \cdot x \in \forall P$ as required.

Logical operations in $(\mathsf{Ren}, Pred)$ are given by their standard meanings in classical set theory. Propositions are elements of $\mathbf{O}$ rather than of $\{0, 1\}$. Note that the axiom

of unique choice (which identifies functional relations with morphisms) is not valid in $(\mathsf{Ren}, \mathit{Pred})$. Indeed, a functional relation, i.e., a predicate P on $\mathbf{X} \times \mathbf{Y}$ such that for all $x \in |\mathbf{X}|$ there is a unique $y \in |\mathbf{Y}|$ such that $(x, y) \in P$, does not in general give rise to a morphism from $\mathbf{X}$ to $\mathbf{Y}$. Indeed, in [16] it was shown that the Axiom of Unique Choice is an inconsistent extension of the theory of contexts. There is a unique function f such that $(x, f(x)) \in P$ for all $x \in |\mathbf{X}|$ but in general it will be invariant under permutations, not all renamings.

7 Conclusions

Nominal renaming sets are a natural evolution of nominal sets. The connection between Ren and the category of nominal sets [13] is evident. However, the details are not entirely straightforward; proofs have to be changed and sometimes they take on a very different character. In particular the definition of the action for function spaces in the permutative case $(\sigma \cdot f)(x) = \sigma^{-1} \cdot f(\sigma \cdot x)$ does not carry over, because σ^{-1} need not exist. It is replaced by an *a priori* partial definition (Definition 20) which is totalised with a unique extension theorem (Lemma 19).

Nominal renaming sets are related to $\mathsf{Set}^{\mathbb{F}}$ as featured in [7] and [16]; they correspond to the pullback-preserving presheaves (Theorem 38). Ren has appeared before; the first author discussed the idea in his thesis [9, Section 11.1] and studied a generalisation of nominal renaming sets as an algebraic theory over nominal sets in [12]. Meanwhile, a category equivalent to Ren was independently presented, also as an algebraic theory over nominal sets, in [8, Definition 2.4]. The constructions presented in this paper, notably the exponential and the tripos structure, are new and do not follow directly from these earlier results. It is thanks to these explicit constructions that Ren can serve as a sets-based semantic underpinning for weak HOAS, i.e. that we can do calculations entirely within Ren, without switching to a functor category. To our knowledge, we are the first to suggest applying renaming sets as a semantic basis for weak HOAS in the sense of Despeyroux [5] and in particular the theory of contexts [17].

Future work.

It will be interesting to give details of a proof of validity of the theory of contexts using nominal renaming sets. In this conference paper we have not done that, but such a proof could be based on the one in [17], and should be simplified thanks to the use of set-theoretic language (most of the time). Note that no notion of forcing is required.

Nominal sets have been generalised to 'nominal domains' [22] thus giving access to fresh names in denotational semantics. It would be interesting to generalise renaming sets in this way as well, giving access to fresh names with substitution within a functional metalanguage. Here, our explicitation of the exponential (Section 3) may be particularly useful.

Nominal renaming sets belong to a family of structures with a finitely supported substitution action. Probably they are the simplest, but others may also be interesting. For example, define a *nominal substitution set* to be a pair of a nominal renaming set $\mathbf{X}$ and a function $\mathit{sub} : ((\mathbf{A} \Rightarrow \mathbf{X}) \times \mathbf{X}) \Rightarrow \mathbf{X}$ such that

– If $a \# z$ then $z[a \mapsto x] = z$.
– If $a \# y$ then $z[a \mapsto x][b \mapsto y] = z[b \mapsto y][a \mapsto x[b \mapsto y]]$.

Here we write $z[a{\mapsto}x]$ for $sub(\lambda y{\in}\mathbf{A}.[a{\mapsto}y]\cdot z,\ x)$. A category of nominal substitution sets has arrows maps $F : \mathbf{X} \longrightarrow \mathbf{Y}$ such that $F(z[a{\mapsto}x]) = (F(z))[a{\mapsto}F(x)]$.

(This can also be phrased directly using 'normal' sets.) We conjecture that this definition or a refinement of it will be useful to give semantics to typings like $(tm \to tm) \to tm$ used for binders, for example in Twelf [19], for which hitherto only purely presheaf semantics are available [16].

References

1. Benton, N., Leperchey, B.: Relational reasoning in a nominal semantics for storage. In: Urzyczyn, P. (ed.) TLCA 2005. LNCS, vol. 3461, pp. 86–101. Springer, Heidelberg (2005)
2. Brunner, N.: 75 years of independence proofs by Fraenkel-Mostowski permutation models. Mathematica Japonica 43, 177–199 (1996)
3. Bucalo, A., Honsell, F., Miculan, M., Scagnetto, I., Hofmann, M.: Consistency of the theory of contexts. Journal of Functional Programming 16(3), 327–395 (2006)
4. Despeyroux, J.: A higher-order specification of the π–calculus. In: IFIP TCS, pp. 425–439 (2000)
5. Despeyroux, J., Felty, A.P., Hirschowitz, A.: Higher-order abstract syntax in COQ. In: Urzyczyn, P. (ed.) TLCA 2005. LNCS, vol. 3461, pp. 124–138. Springer, Heidelberg (2005)
6. Despeyroux, J., Hirschowitz, A.: Higher-order abstract syntax with induction in COQ. In: Pfenning, F. (ed.) LPAR 1994. LNCS, vol. 822, pp. 159–173. Springer, Heidelberg (1994)
7. Fiore, M.P., Plotkin, G.D., Turi, D.: Abstract syntax and variable binding. In: LICS 1999, pp. 193–202. IEEE, Los Alamitos (1999)
8. Fiore, M.P., Staton, S.: A congruence rule format for name-passing process calculi from mathematical structural operational semantics. In: LICS 2006, pp. 49–58. IEEE, Los Alamitos (2006)
9. Gabbay, M.J.: A Theory of Inductive Definitions with alpha-Equivalence. PhD thesis, Cambridge, UK (2000)
10. Gabbay, M.J.: A General Mathematics of Names. Information and Computation 205, 982–1011 (2007)
11. Gabbay, M.J.: Nominal renaming sets. Technical Report HW-MACS-TR-0058, Heriot-Watt University (2007), http://www.gabbay.org.uk/papers.html#nomrs-tr
12. Gabbay, M.J., Mathijssen, A.: Capture-avoiding Substitution as a Nominal Algebra. In: Barkaoui, K., Cavalcanti, A., Cerone, A. (eds.) ICTAC 2006. LNCS, vol. 4281, pp. 198–212. Springer, Heidelberg (2006)
13. Gabbay, M.J., Pitts, A.M.: A New Approach to Abstract Syntax with Variable Binding (journal version). Formal Aspects of Computing 13(3–5), 341–363 (2001)
14. Gabbay, M.J., Pitts, A.M.: A new approach to abstract syntax involving binders. In: 14th Annual Symposium on Logic in Computer Science, pp. 214–224. IEEE Computer Society Press, Los Alamitos (1999)
15. Hirschkoff, D.: A full formalization of pi-calculus theory in the Calculus of Constructions. In: Gunter, E.L., Felty, A.P. (eds.) TPHOLs 1997. LNCS, vol. 1275, pp. 153–169. Springer, Heidelberg (1997)
16. Hofmann, M.: Semantical analysis of higher-order abstract syntax. In: 14th Annual Symposium on Logic in Computer Science, pp. 204–213. IEEE, Los Alamitos (1999)
17. Honsell, F., Miculan, M., Scagnetto, I.: An axiomatic approach to metareasoning on nominal algebras in HOAS. In: Orejas, F., Spirakis, P.G., van Leeuwen, J. (eds.) ICALP 2001. LNCS, vol. 2076, pp. 963–978. Springer, Heidelberg (2001)

18. McKinna, J., Pollack, R.: Some lambda calculus and type theory formalized. Journal of Automated Reasoning 23(3-4), 373–409 (1999)
19. Pfenning, F., Schürmann, C.: System description: Twelf - a meta-logical framework for deductive systems. In: Ganzinger, H. (ed.) CADE 1999. LNCS, vol. 1632, pp. 202–206. Springer, Heidelberg (1999)
20. Shinwell, M.R.: The Fresh Approach: Functional Programming with Names and Binders. PhD thesis, Computer Laboratory, University of Cambridge (December 2004)
21. Shinwell, M.R., Pitts, A.M., Gabbay, M.J.: FreshML: Programming with binders made simple. In: ICFP 2003. SIGPLAN Not., vol. 38(9), pp. 263–274. ACM Press, New York (2003)
22. Shinwell, M.R., Pitts, A.M.: On a monadic semantics for freshness. Theoretical Computer Science 342(1), 28–55 (2005)
23. Shinwell, M.R., Pitts, A.M.: Fresh objective Caml user manual. Technical Report UCAM-CL-TR-621, University of Cambridge (2005)
24. Staton, S.: Name-passing process calculi: operational models and structural operational semantics. PhD thesis, University of Cambridge (2007)
25. Urban, C., Tasson, C.: Nominal techniques in Isabelle/HOL. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS, vol. 3632, pp. 38–53. Springer, Heidelberg (2005)

Imogen: Focusing the Polarized Inverse Method for Intuitionistic Propositional Logic

Sean McLaughlin and Frank Pfenning

Department of Computer Science
Carnegie Mellon University

Abstract. In this paper we describe Imogen, a theorem prover for intuitionistic propositional logic using the focused inverse method. We represent fine-grained control of the search behavior by *polarizing* the input formula. In manipulating the polarity of atoms and subformulas, we can often improve the search time by several orders of magnitude. We tested our method against seven other systems on the propositional fragment of the ILTP library. We found that our prover outperforms all other provers on a substantial subset of the library.

1 Introduction

Imogen is a theorem prover for intuitionistic propositional logic (IPL) based on a focused inverse method with explicit polarities. The *inverse method* [15, 7] uses forward saturation, generalizing resolution to non-classical logics. Focusing [1, 14] reduces the search space in a sequent calculus by restricting the application of inference rules based on the *polarities* of the connectives and atomic formulas. One of the novel aspects of Imogen is that it exploits inherent flexibility in the assignment of polarities to subformulas to optimize proof search. Different assignments of polarities can yield dramatically different performance.

Raths and Otten [18] compare seven systems on the ILTP library [19], a collection of challenge problems for intuitionistic logic provers. In contrast to Imogen, all these use backward search in a contraction-free sequent calculus. This difference in basic approach is reflected in a unique performance profile. Imogen clearly outperforms the other provers on an interesting subset of the benchmark problems, with a tendency to do better on non-theorems. Some problems that appear difficult for backward search are solved almost instantaneously by Imogen, and vice versa. We therefore consider Imogen an interesting and viable alternative for intuitionistic theorem proving.

In this system description we give an overview of the basic principles underlying Imogen, its implementation, and analyze its performance compared to other provers for IPL. The theoretical foundations for Imogen are mostly contained in published papers cited in this description; we therefore do not explicitly state or prove any metatheorems. The source code for Imogen is available at `http://www.cs.cmu.edu/~seanmcl/Imogen`.

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 174–181, 2008.

2 The Polarized Inverse Method

In this section we sketch the main principles underlying Imogen and their interaction: focusing, polarization, and the inverse method.

2.1 Focusing

Focusing is a method to restrict the space of possible proofs in a cut-free sequent calculus without affecting provability. It was originally developed for backward search in classical linear logic [1], but has been applied to other non-classical logics [11,14] as well as forward search [5].

Focusing is based on two observations about the properties of connectives. The first is that certain connectives can always be eagerly decomposed during backward proof search without losing completeness. For example, the goal of proving $A \supset B$ can always be decomposed to proving B under additional assumption A. Such connectives are said to have *negative polarity*. As long as the top-level connective stays negative, we can continue the decomposition eagerly without considering any other possibilities. In contrast, for a formula such as $A \vee B$, we have to make a choice whether to try to prove A or B. Such connectives are said to have *positive polarity*. Surprisingly, as long as the top-level connective stays positive, we can continue the decomposition eagerly, making a choice at each step. Moreover, we can arbitrarily assign positive or negative polarity to atomic formulas and restrict the use of atoms in initial sequents.

Proofs that satisfy all three restrictions are called *focused*. Imogen restricts its forward search to *focused proofs*, in a manner explained in the next two sections, drastically reducing its search space when compared to the usual sequent calculus.

2.2 Polarized Formulas

In linear logic, the polarity of each connective is uniquely determined. This is not true for intuitionistic logic where conjunction and truth are inherently ambiguous. We therefore assign polarities to formulas in a preprocessing phase. It is convenient to represent the result as a *polarized formula* [12] where immediately nested formulas always have the same polarity, unless an explicit polarity-shifting connective $\uparrow$ or $\downarrow$ is encountered. These coercions are called *shifts*.

Implication has slightly special status, in that its left-hand side has opposite polarity from its right-hand side. This is because in the sequent calculus for intuitionistic logic, the focusing behavior of connectives on the left-hand side is the opposite of their behavior on the right-hand side. (Here the meta-variable P ranges over atomic propositions.)

$$\text{Positive formulas } A^+ ::= P^+ \mid A^+ \vee A^+ \mid \bot \mid A^+ \wedge A^+ \mid \top \mid \downarrow A^-$$

$$\text{Negative formulas } A^- ::= P^- \mid A^+ \supset A^- \mid A^- \mathbin{\bar{\wedge}} A^- \mid \bar{\top} \mid \uparrow A^+$$

The translation A^- of an (unpolarized) formula F in IPL is nondeterministic, subject only to the constraint that the translation $|A^-| = F$.

$$
\begin{array}{lll}
|A^+ \vee B^+| = |A^+| \vee |B^+| & |\bot| = \bot & |P^+| = P \\
|A^+ \wedge B^+| = |A^+| \wedge |B^+| & |\top| = \top & |{\downarrow}A^-| = |A^-| \\
|A^- \mathbin{\overline{\wedge}} B^-| = |A^-| \wedge |B^-| & |\overline{\top}| = \top & |P^-| = P \\
|A^+ \supset B^-| = |A^+| \supset |B^-| & |{\uparrow}A^+| = |A^+| &
\end{array}
$$

For example, the formula $((A \vee C) \wedge (B \supset C)) \supset (A \supset B) \supset C$ can be interpreted as any of the following polarized formulas (among others):

$$
\begin{array}{l}
(({\downarrow}A^- \vee {\downarrow}C^-) \wedge {\downarrow}({\downarrow}B^- \supset C^-)) \supset ({\downarrow}({\downarrow}A^- \supset B^-) \supset C^-) \\
{\downarrow}{\uparrow}(({\downarrow}A^- \vee {\downarrow}C^-) \wedge {\downarrow}({\downarrow}B^- \supset C^-)) \supset ({\downarrow}{\uparrow}{\downarrow}({\downarrow}A^- \supset B^-) \supset C^-) \\
{\downarrow}({\uparrow}(A^+ \vee C^+) \mathbin{\overline{\wedge}} (B^+ \supset {\uparrow}C^+)) \supset ({\downarrow}(A^+ \supset {\uparrow}B^+) \supset {\uparrow}C^+)
\end{array}
$$

Shift operators have highest binding precedence in our presentation of the examples. As we will see, the choice of translation determines the search behavior on the resulting polarized formula. Different choices can lead to search spaces with radically different structure [6].

2.3 From Focused Proofs to Big-Step Inferences

A sequent of intuitionistic logic has the form $\Gamma \Longrightarrow A$, where Γ is a set or multiset of formulas. For purposes of Imogen it is convenient to always maintain Γ as a set, without duplicates. Since we can always eagerly decompose negative connectives on the right of a sequent and positive connectives on the left, the only sequents in our polarized calculus we need to consider have negative formulas on the left or positive formulas on the right, in addition to atoms which can appear with either polarity on either side. The right-hand side could also be empty if we are deriving a contradiction. We call such sequents *stable*.

$$
\begin{array}{ll}
\text{Stable Hypotheses} & \Gamma ::= \cdot \mid \Gamma, A^- \mid \Gamma, P^+ \\
\text{Stable Conclusions} & \gamma ::= A^+ \mid P^- \mid \cdot \\
\text{Stable Sequents} & \Gamma \Longrightarrow \gamma
\end{array}
$$

We exploit focusing on polarized formulas to derive big-step rules that go from stable sequents as premises to stable sequents as conclusions. Completeness of focusing tells us that these derived rules, by themselves, are sufficient to prove all valid stable sequents. Rather than formally specify this rule generation (see, for example, Andreoli [2] for the linear case), we only illustrate the process, continuing with the example above.

$$(({\downarrow}A^- \vee {\downarrow}C^-) \wedge {\downarrow}({\downarrow}B^- \supset C^-)) \supset ({\downarrow}({\downarrow}A^- \supset B^-) \supset C^-)$$

The input formula is always translated to a negative formula, which we break down to a set of stable sequents by applying invertible rules. Here we obtain the two sequents

$$
\begin{array}{l}
A, {\downarrow}B \supset C, {\downarrow}A \supset B \Longrightarrow C \\
C, {\downarrow}B \supset C, {\downarrow}A \supset B \Longrightarrow C
\end{array}
$$

We search for proofs of these two stable sequents independently. For each stable sequent, we focus on each constituent formula in turn, and decompose it until we reach

all stable sequents as premises. Each possibility yields a new big-step inference rule. We continue to analyze its premises recursively in the same manner. As an example, we show the process for the first goal above.

Focusing on A yields the initial sequent $A \Longrightarrow A$. Focusing on $\downarrow B \supset C$ and $\downarrow A \supset B$ yield the big-step rules

$$\frac{\Gamma \Longrightarrow B}{\Gamma, \downarrow B \supset C \Longrightarrow C} \qquad \frac{\Gamma \Longrightarrow A}{\Gamma, \downarrow A \supset B \Longrightarrow B}$$

2.4 The Inverse Method with Big-Step Rules

The usual (small-step) inverse method applies sequent calculus rules in the forward direction so that each derived formula is a subformula of the original goal. The subformula property is already built into the generation of the rules, so all we need to do now is to apply the big-step rules to saturation in the forward direction. To start the process, each derived rule with no premises is considered as an initial sequent.

To prove the first stable sequent in our example, we begin with the initial sequent $A \Longrightarrow A$. We only have two inference rules, of which only the second applies. The application of this rule derives the new fact $A, \downarrow A \supset B \Longrightarrow B$. Once again, we have only one choice: applying the first rule to this new sequent. The application yields $A, \downarrow A \supset B, \downarrow B \supset C \Longrightarrow C$ which is our goal.

In general, forward inference may only generate a strengthened form of the goal sequent, so we need check if any derived sequents subsume the goal. $\Gamma \Longrightarrow \gamma$ *subsumes* $\Gamma' \Longrightarrow \gamma'$ if $\Gamma \subseteq \Gamma'$ and $\gamma \subseteq \gamma'$. The inference process *saturates* if any new sequent we can derive is already subsumed by a previously derived sequent. If none of these subsume the goal sequent, the goal is not provable and we explicitly fail. In this case, the saturated database may be considered a kind of countermodel for the goal sequent. If the goal sequent is found, Imogen can reconstruct a natural deduction proof term as a witness to the formula's validity.

3 Optimizations and Heuristics

A problem with focusing becomes apparent when considering formulas such as

$$A = (A_1 \vee B_1) \wedge (A_2 \vee B_2) \wedge \cdots \wedge (A_n \vee B_n)$$

Focusing on A on the right will produce 2^n inference rules. Inverting F on the left will produce a single rule with 2^n premises. To avoid exponential behavior such as this, we can change the polarities of the subformulas by adding *double shifts*, $\downarrow\uparrow$ and $\uparrow\downarrow$:

$$A' = \downarrow\uparrow(A_1 \vee B_1) \wedge \downarrow\uparrow(A_2 \vee B_2) \wedge \cdots \wedge \downarrow\uparrow(A_n \vee B_n).$$

The double shifts break the focusing and inversion phases respectively, leading to a linear number of rules and premises at the expense of an increased number of inverse method deductions. In the extreme, if we insert double shifts before every subformula, we can emulate the inverse method for the ordinary sequent calculus. Imogen currently uses heuristics to insert double shifts for avoiding an exponential explosion.

Imogen first translates to polarized form by a simple method that inserts the fewest shifts, making the choice of conjunction accordingly. Using additional heuristics, Imogen may modify this decision, adding shifts and swapping conjunction and atom polarities to improve the search behavior. Sometimes this leads to a very different search space for problems that are syntactically very similar[1]. Roughly, we count the number of rules and premises that will result from focusing. If such numbers are very large with respect to the input formula, we insert double shifts at a subformula of the goal that is causing a part of the explosion and check the number of rules and premises on the new formula. We continue in this way until a "reasonable" number of premises and rules is reached.

We maintain the hypotheses as sets. Thus, contraction is handled automatically by the application of multi-premise rules.

Formulas which appear in a stable goal sequent will appear in *every* sequent which backward search could construct and are therefore redundant. We omit such *global* assumptions from all sequents. Another helpful optimization is *backward subsumption*. When a new sequent is derived, we remove all sequents that it subsumes from the database. These effects are quantified in section 5.

4 Inference Engine

Imogen's saturation algorithm is based on the Otter loop [16]. It maintains Otter's two distinct databases for *active* sequents[2], those sequents that have had all inference rules applied to them, and *kept* sequents that have not yet been considered for inferences. New rules are generated when a multiple premise rule is matched against an active sequent. This method of matching multi-premise rules incrementally is called *partially applied rule generation*.

The algorithm proceeds as follows. It first polarizes the input formula and runs an initial stabilization pass to determine the stable sequents to prove. The initial sequents and derived rules are then generated using focusing. As an optimization, subformulas are given unique labels to allow fast formula comparison. The final step before search is to initialize the kept sequent database with the initial sequents.

At this stage, Imogen begins the forward search. It selects a kept sequent based on some fair strategy. The sequent is matched against the first premise of all current rules. The matching process will produce new sequents that are put into the kept database, as well as new partially applied rules. The new rules are recursively matched against the active database, and the resulting sequents are put into the kept database. This process repeats until either the kept database becomes empty, in which case the search space is saturated and the formula is invalid, or until the goal sequent is subsumed by a derived sequent.

5 Evaluation

We evaluated our prover on the propositional fragment of the ILTP [19, version 1.1.2] library of problems for intuitionistic theorem provers. The 274 problems are divided

[1] This effect can be seen in the erratic results of problem class SYJ206 in section 5.

[2] Sometimes called the "set of support".

into 12 families of difficult problems such as the pigeonhole principle, labeled SYJ201 to SYJ212. For each family, there are 20 instances of increasing size. There are also 34 miscellaneous problems. The provers that are currently evaluated are ft-C [20, version 1.23], ft-Prolog [20, version 1.23], LJT [8], PITP [3, version 3.0], PITPINV [3, version 3.0], and STRIP [13, version 1.1]. These provers represent a number of different methods of theorem proving in IPL, yet forward reasoning is conspicuously absent. Imogen solved 261 of the problems. PITPINV was the only prover to solve more. Some illustrative examples of difficult problems are shown in the following table:

Prover	ft-Prolog	ft-C	LJT	PITP	PITPINV	IPTP	STRIP	Imogen
Solved (out of 274)	188	199	175	238	262	209	205	261
SYN007+1.014	-0.01	-0.01	stack	large	large	large	alloc	-0.1
SYJ201+1.018	0.28	0.04	0.4	0.01	0.01	2.31	0.23	25.5
SYJ201+1.019	0.36	0.04	0.47	0.01	0.01	2.82	0.32	28.0
SYJ201+1.020	0.37	0.05	0.55	0.01	0.01	3.47	0.34	28.35
SYJ202+1.007	516.55	76.3	memory	0.34	0.31	13.38	268.59	64.6
SYJ202+1.008	time	time	memory	3.85	3.47	97.33	time	time
SYJ202+1.009	time	time	memory	50.25	42.68	time	time	time
SYJ202+1.010	time	time	memory	time	time	time	time	time
SYJ205+1.018	time	time	0.01	0.01	7.49	0.09	time	0.01
SYJ205+1.019	time	time	0.01	0.01	15.89	0.09	time	0.01
SYJ205+1.020	time	time	0.01	0.01	33.45	0.1	time	0.01
SYJ206+1.018	time	time	memory	1.01	0.96	9.01	8.18	56.2
SYJ206+1.019	time	time	memory	1.95	1.93	18.22	14.58	394.14
SYJ206+1.020	time	time	memory	3.92	3.89	36.35	33.24	42.7
SYJ207+1.018	time	time	time	time	-68.71	time	time	-42.6
SYJ207+1.019	time	time	time	time	-145.85	time	time	-63.6
SYJ207+1.020	time	time	time	time	-305.21	time	time	-97.25
SYJ208+1.018	time	time	memory	-0.99	-0.95	time	time	-184.14
SYJ208+1.019	time	time	memory	-1.36	-1.35	memory	mem	-314.31
SYJ208+1.020	time	time	memory	-1.76	-1.80	memory	mem	-506.02
SYJ209+1.018	time	time	time	time	-13.44	time	time	-0.01
SYJ209+1.019	time	time	time	time	-28.68	time	time	-0.01
SYJ209+1.020	time	time	time	time	-60.54	time	time	-0.02
SYJ211+1.018	time	time	time	-43.65	-31.51	time	time	-0.02
SYJ211+1.019	time	time	time	-91.75	-66.58	time	time	-0.02
SYJ211+1.020	time	time	time	-191.57	-139.67	time	time	-0.02
SYJ212+1.018	-0.01	-0.01	memory	-1.31	-1.37	time	-8.5	-0.02
SYJ212+1.019	-0.01	-0.01	memory	-2.7	-2.75	time	-17.41	-0.03
SYJ212+1.020	-0.01	-0.01	memory	-5.51	-5.51	time	-38.94	-0.04

The table uses the notation of [18]. All times are in seconds. The entry "memory" indicates that the prover process ran out of memory. A "time" entry indicates that the prover was unable to solve the problem within the ten minute time limit. A negative number indicates the time to ascertain that a formula is *not* valid. All statistics except for those of Imogen were executed on a 3.4 GHz Xeon processor running Linux [18]. The Imogen statistics are a 2.4 GHz Intel Core 2 Duo on Mac OS X. Thus the Imogen statistics are conservative.

6 Conclusion

The most closely related system to Imogen is Linprover [4], which is an inverse method prover for intuitionistic linear logic exploiting focusing, but not polarization. We do not explicitly compare our results to Linprover, which incurs additional overhead due to the necessary maintenance of linearity constraints. We are also aware of two provers for first-order intuitionistic logic based on the inverse method, Gandalf [21] and Sandstorm [9], both of which partially exploit focusing. We do not compare Imogen to these either, since they incur substantial overhead due to unification, contraction, and more complex subsumption.

Imogen could be improved in a number of ways. Selecting sequents from the kept database for rule application could be improved by a more intelligent ordering of formulas. Better heuristics for assigning polarities to subformulas, especially atoms, seem to offer the biggest source of performance gains. Experimenting with double shifts and atom polarities by hand greatly increased performance, but as yet we have no sophisticated methods for determining more optimal assignments of polarities.

We implemented Imogen with the eventual goal to generalize the system to first-order intuitionistic logic and logical frameworks and were somewhat surprised that even a relatively straightforward implementation in a high-level language is not only competitive with previous provers based on backward search, but clearly better on a significant portion of accepted benchmark problems. We plan to begin experiments using the polarized inverse method for LF [10] and M2, the metalogic of Twelf [17].

One of Imogen's strengths is its ability to do redundancy elimination. The databases can grow large, making deriving further inferences slower. Yet when a strong sequent is derived, it is not uncommon for half or more of the database to be subsumed and eliminated with backward subsumption, thus allowing Imogen to continue making deductions at a much higher rate. We believe that this will be important in solving difficult problems in more complex logics.

Our experience with Imogen, in addition to the evidence provided by the provers cited above, adds strength to our thesis that the polarized inverse method works well on non-classical logics of different kinds.

Acknowledgments

We have been generously supported in this work by NSF grant CCR-0325808. We would like to thank Kaustuv Chaudhuri for helpful discussions regarding Imogen. We also thank the anonymous reviewers for helpful suggestions and correcting some minor errors.

References

1. Andreoli, J.-M.: Logic programming with focusing proofs in linear logic. Journal of Logic and Computation 2(3), 297–347 (1992)
2. Andreoli, J.-M.: Focussing and proof construction. Annals of Pure and Applied Logic 107(1–3), 131–163 (2001)

3. Avellone, A., Fiorino, G., Moscato, U.: A new O(n log n)-space decision procedure for propositional intuitionistic logic. In: Kurt Goedel Society Collegium Logicum, vol. VIII, pp. 17–33 (2004)
4. Chaudhuri, K.: The Focused Inverse Method for Linear Logic. PhD thesis, Carnegie Mellon University, Technical report CMU-CS-06-162 (December 2006)
5. Chaudhuri, K., Pfenning, F.: Focusing the inverse method for linear logic. In: Ong, L. (ed.) Computer Science Logic, pp. 200–215. Springer, Heidelberg (2005)
6. Chaudhuri, K., Pfenning, F., Price, G.: A logical characterization of forward and backward chaining in the inverse method. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS, vol. 4130, pp. 97–111. Springer, Heidelberg (2006)
7. Degtyarev, A., Voronkov, A.: The inverse method. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, ch. 4, vol. I, pp. 179–272. Elsevier Science, Amsterdam (2001)
8. Dyckhoff, R.: Contraction-free sequent calculi for intuitionistic logic. Journal of Symbolic Logic 57(3), 795–807 (1992)
9. Garg, D., Murphy, T., Price, G., Reed, J., Zeilberger, N.: Team red: The Sandstorm theorem prover, http://www.cs.cmu.edu/~tom7/papers/
10. Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. Journal of the ACM 40(1), 143–184 (1993)
11. Howe, J.M.: Proof Search Issues in Some Non-Classical Logics. PhD thesis, University of St. Andrews, Scotland (1998)
12. Lamarche, F.: Games semantics for full propositional linear logic. In: Proceedings of the 10th Annual Symposium on Logic in Computer Science (LICS 1995), San Diego, California, pp. 464–473. IEEE Computer Society, Los Alamitos (1995)
13. Larchey-Wendling, D., Méry, D., Galmiche, D.: STRIP: Structural sharing for efficient proof-search. In: Goré, R.P., Leitsch, A., Nipkow, T. (eds.) IJCAR 2001. LNCS, vol. 2083, pp. 670–674. Springer, Heidelberg (2001)
14. Liang, C., Miller, D.: Focusing and polarization in intuitionistic logic. In: Duparc, J., Henzinger, T.A. (eds.) CSL 2007. LNCS, vol. 4646, pp. 451–465. Springer, Heidelberg (2007)
15. Maslov, S.Y.: An inverse method for establishing deducibility in classical predicate calculus. Doklady Akademii nauk SSSR 159, 17–20 (1964)
16. McCune, W.W.: OTTER 3.0 reference manual and guide. Technical Report ANL-94/6, Argonne National Laboratory/IL, USA (1994)
17. Pfenning, F., Schürmann, C.: System description: Twelf - A meta-logical framework for deductive systems. In: Ganzinger, H. (ed.) CADE 1999. LNCS, vol. 1632, pp. 202–206. Springer, Heidelberg (1999)
18. Raths, T., Otten, J.: The ILTP Library, http://www.iltp.de/
19. Raths, T., Otten, J., Kreitz, C.: The ILTP problem library for intuitionistic logic. J. Autom. Reasoning 38(1-3), 261–271 (2007)
20. Sahlin, D., Franzén, T., Haridi, S.: An intuitionistic predicate logic theorem prover. Journal of Logic and Computation 2(5), 619–656 (1992)
21. Tammet, T.: A resolution theorem prover for intuitionistic logic. In: McRobbie, M.A., Slaney, J.K. (eds.) CADE 1996. LNCS, vol. 1104, pp. 2–16. Springer, Heidelberg (1996)

Model Checking – My 27-Year Quest to Overcome the State Explosion Problem

Edmund M. Clarke

Computer Science Department
Carnegie Mellon University

Abstract. Model Checking is an automatic verification technique for state-transition systems that are finite-state or that have finite-state abstractions. In the early 1980's in a series of joint papers with my graduate students E.A. Emerson and A.P. Sistla, we proposed that Model Checking could be used for verifying concurrent systems and gave algorithms for this purpose. At roughly the same time, Joseph Sifakis and his student J.P. Queille at the University of Grenoble independently developed a similar technique. Model Checking has been used successfully to reason about computer hardware and communication protocols and is beginning to be used for verifying computer software. Specifications are written in temporal logic, which is particularly valuable for expressing concurrency properties. An intelligent, exhaustive search is used to determine if the specification is true or not. If the specification is not true, the Model Checker will produce a counterexample execution trace that shows why the specification does not hold. This feature is extremely useful for finding obscure errors in complex systems. The main disadvantage of Model Checking is the state-explosion problem, which can occur if the system under verification has many processes or complex data structures. Although the state-explosion problem is inevitable in worst case, over the past 27 years considerable progress has been made on the problem for certain classes of state-transition systems that occur often in practice. In this talk, I will describe what Model Checking is, how it works, and the main techniques that have been developed for combating the state explosion problem.

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, p. 182, 2008.

On the Relative Succinctness of Nondeterministic Büchi and co-Büchi Word Automata

Benjamin Aminof, Orna Kupferman, and Omer Lev

Hebrew University, School of Engineering and Computer Science, Jerusalem 91904, Israel
{benj,orna}@cs.huji.ac.il, omerl@math.huji.ac.il

Abstract. The practical importance of automata on infinite objects has motivated a re-examination of the complexity of automata-theoretic constructions. One such construction is the translation, when possible, of nondeterministic Büchi word automata (NBW) to nondeterministic co-Büchi word automata (NCW). Among other applications, it is used in the translation (when possible) of LTL to the alternation-free μ-calculus. The best known upper bound for the translation of NBW to NCW is exponential (given an NBW with n states, the best translation yields an equivalent NCW with $2^{O(n \log n)}$ states). On the other hand, the best known lower bound is trivial (no NBW with n states whose equivalent NCW requires even $n+1$ states is known). In fact, only recently was it shown that there is an NBW whose equivalent NCW requires a different structure.

In this paper we improve the lower bound by showing that for every integer $k \geq 1$ there is a language L_k over a two-letter alphabet, such that L_k can be recognized by an NBW with $2k+1$ states, whereas the minimal NCW that recognizes L_k has $3k$ states. Even though this gap is not asymptotically very significant, it nonetheless demonstrates for the first time that NBWs are more succinct than NCWs. In addition, our proof points to a conceptual advantage of the Büchi condition: an NBW can abstract precise counting by counting to infinity with two states. To complete the picture, we consider also the reverse NCW to NBW translation, and show that the known upper bound, which duplicates the state space, is tight.

1 Introduction

Finite *automata on infinite objects* were first introduced in the 60's, and were the key to the solution of several fundamental decision problems in mathematics and logic [3,13,16]. Today, automata on infinite objects are used for *specification* and *verification* of nonterminating systems. The automata-theoretic approach to verification views questions about systems and their specifications as questions about languages, and reduces them to automata-theoretic problems like containment and emptiness [11,21]. Recent industrial-strength property-specification languages such as Sugar [2], ForSpec [1], and the recent standard PSL 1.01 [5] include regular expressions and/or automata, making specification and verification tools that are based on automata even more essential and popular.

There are many ways to classify an automaton on infinite words. One is the type of its acceptance condition. For example, in *Büchi* automata, some of the states are designated as accepting states, and a run is accepting iff it visits states from the accepting set

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 183–197, 2008.

infinitely often [3]. Dually, in *co-Büchi* automata, a run is accepting iff it visits states from the accepting set only finitely often. Another way to classify an automaton is by the type of its branching mode. In a *deterministic* automaton, the transition function maps the current state and input letter to a single successor state. When the branching mode is *nondeterministic*, the transition function maps the current state and letter to a set of possible successor states. Thus, while a deterministic automaton has at most a single run on an input word, a nondeterministic automaton may have several runs on an input word, and the word is accepted by the automaton if at least one of the runs is accepting.

Early automata-based algorithms aimed at showing decidability. The complexity of the algorithm was not of much interest. Things have changed in the early 80's, when decidability of highly expressive logics became of practical importance in areas such as artificial intelligence and formal reasoning about systems. The change was reflected in the development of two research directions: (1) direct and efficient translations of logics to automata [23,19,20], and (2) improved algorithms and constructions for automata on infinite objects [18,4,15]. For many problems and constructions, our community was able to come up with satisfactory solutions, in the sense that the upper bound (the complexity of the best algorithm or the blow-up in the best known construction) coincides with the lower bound (the complexity class in which the problem is hard, or the blow-up that is known to be unavoidable). For some problems and constructions, however, the gap between the upper bound and the lower bound is significant. This situation is especially frustrating, as it implies that not only we may be using algorithms that can be significantly improved, but also that something is missing in our understanding of automata on infinite objects.

One such problem, which this article studies, is the problem of translating, when possible, a nondeterministic Büchi word automaton (NBW) to an equivalent nondeterministic co-Büchi word automaton (NCW). NCWs are less expressive than NBWs. For example, the language $\{w : w \text{ has infinitely many } a\text{'s}\}$ over the alphabet $\{a, b\}$ cannot be recognized by an NCW. The best translation of an NBW to an NCW (when possible) that is currently known actually results in a deterministic co-Büchi automaton (DCW), and it goes via an intermediate deterministic Streett automaton. The determinization step involves an exponential blowup in the number of states [18]. Hence, starting with an NBW with n states, we end up with a DCW with $2^{O(n \log n)}$ states.

The exponential upper bound is particularly annoying, since the best known lower bound is trivial. That is, no NBW with n states whose equivalent NCW requires even $n + 1$ states is known. In fact, only recently was it shown that there is an NBW whose equivalent NCW requires a different structure [8]. Beyond the theoretical challenge in closing the exponential gap, and the fact it is related to other exponential gaps in our knowledge [7], the translation of NBW to NCW has immediate applications in symbolic LTL model checking. We elaborate on this point below.

It is shown in [9] that given an LTL formula ψ, there is an alternation-free μ-calculus (AFMC) formula equivalent to $\forall\psi$ iff ψ can be recognized by a deterministic Büchi automaton (DBW). Evaluating specifications in the alternation-free fragment of μ-calculus can be done with linearly many symbolic steps. In contrast, direct LTL model checking reduces to a search for bad-cycles, whose symbolic implementation involves

nested fixed-points, and is typically quadratic [17]. The best known translations of LTL to AFMC first translates the LTL formula ψ to a DBW, which is then linearly translated to an AFMC formula for $\forall\psi$. The translation of LTL to DBW, however, is doubly-exponential, thus the overall translation is doubly-exponential, with only an exponential matching lower bound [9]. A promising direction for coping with this situation was suggested in [9]: Instead of translating the LTL formula ψ to a DBW, one can translate $\neg\psi$ to an NCW. This can be done either directly, or by translating the NBW for $\neg\psi$ to an equivalent NCW. Then, the NCW can be linearly translated to an AFMC formula for $\exists\neg\psi$, whose negation is equivalent to $\forall\psi$. Thus, a polynomial translation of NBW to NCW would imply a singly-exponential translation of LTL to AFMC.[1]

The main challenge in proving a non-trivial lower bound for the translation of NBW to NCW is the expressiveness superiority of NBW with respect to NCW. Indeed, a language that is a candidate for proving a lower bound for this translation has to strike a delicate balance: the languages has to somehow take advantage of the Büchi acceptance condition, and still be recognizable by a co-Büchi automaton. In particular, attempts to use the main feature of the Büchi condition, namely its ability to easily track infinitely many occurrences of an event, are almost guaranteed to fail, as a co-Büchi automaton cannot recognize languages that are based on such a tracking. Thus, a candidate language has to use the ability of the Büchi condition to easily track the infinity in some subtle way.

In this paper we point to such a subtle way and provide the first non-trivial lower bound for the translation of NBW to NCW. We show that for every integer $k \geq 1$, there is a language L_k over a two-letter alphabet, such that L_k can be recognized by an NBW with $2k + 1$ states, whereas the minimal NCW that recognizes L_k has $3k$ states. Even though this gap is not asymptotically very significant, it demonstrates for the first time that NBWs are more succinct than NCWs. In addition, our proof points to a conceptual advantage of the Büchi condition: an NBW can abstract precise counting by counting to infinity with two states. To complete the picture, we also study the reverse translation, of NCWs to NBWs. We show that the known upper bound for this translation, which doubles the state space of the NCW, is tight.

2 Preliminaries

2.1 Automata on Infinite Words

Given an alphabet Σ, a *word* over Σ is an infinite sequence $w = \sigma_1 \cdot \sigma_2 \cdots$ of letters in Σ. An *automaton* is a tuple $\mathcal{A} = \langle \Sigma, Q, \delta, Q_0, \alpha \rangle$, where Σ is the input alphabet, Q is a finite set of states, $\delta : Q \times \Sigma \to 2^Q$ is a transition function, $Q_0 \subseteq Q$ is a set of initial states, and $\alpha \subseteq Q$ is an acceptance condition. We define several acceptance conditions below. Intuitively, $\delta(q, \sigma)$ is the set of states that $\mathcal{A}$ may move into when it is in the state q and it reads the letter σ. The automaton $\mathcal{A}$ may have several initial states and

[1] Wilke [22] proved an exponential lower-bound for the translation of an NBW for an LTL formula ψ to and AFMC formula equivalent to $\forall\psi$. This lower-bound does not preclude a polynomial upper-bound for the translation of an NBW for $\neg\psi$ to an AFMC formula equivalent to $\exists\neg\psi$, which is our goal.

the transition function may specify many possible transitions for each state and letter, and hence we say that $\mathcal{A}$ is *nondeterministic*. In the case where $|Q_0| = 1$ and for every $q \in Q$ and $\sigma \in \Sigma$, we have that $|\delta(q, \sigma)| \leq 1$, we say that $\mathcal{A}$ is *deterministic*.

Given two states $p, q \in Q$, a *path* of *length* m from p to q is a finite sequence of states $\pi = \pi_0, \pi_1, \cdots, \pi_{m-1}$ such that $\pi_0 = p$, $\pi_{m-1} = q$, and for every $0 \leq i < m-1$, we have that $\pi_{i+1} \in \bigcup_{\sigma \in \Sigma} \delta(\pi_i, \sigma)$. If $\pi_0 \in \bigcup_{\sigma \in \Sigma} \delta(\pi_{m-1}, \sigma)$ then π is a *cycle*. We say that π is *simple* if all the states of π are different. I.e., if for every $1 \leq i < j < m$, we have that $\pi_i \neq \pi_j$. Let $\pi = \pi_0, \pi_1, \cdots, \pi_{m-1}$ be a simple path of length $m \geq k$. The *k-tail* of π is the set $\{\pi_{m-k}, \ldots, \pi_{m-1}\}$ of the last k states of π. Note that since π is simple the size of its k-tail is k.

A run $r = r_0, r_1, \cdots$ of $\mathcal{A}$ on $w = \sigma_1 \cdot \sigma_2 \cdots \in \Sigma^\omega$ is an infinite sequence of states such that $r_0 \in Q_0$, and for every $i \geq 0$, we have that $r_{i+1} \in \delta(r_i, \sigma_{i+1})$. We sometimes refer to runs as words in Q^ω. Note that while a deterministic automaton has at most a single run on an input word, a nondeterministic automaton may have several runs on an input word. Acceptance is defined with respect to the set of states $inf(r)$ that the run r visits infinitely often. Formally, $inf(r) = \{q \in Q \mid \text{ for infinitely many } i \in \mathbb{N}, \text{ we have } r_i = q\}$. As Q is finite, it is guaranteed that $inf(r) \neq \emptyset$. The run r is *accepting* iff the set $inf(r)$ satisfies the acceptance condition α. We consider here the *Büchi* and the *co-Büchi* acceptance conditions. A set $S \subseteq Q$ satisfies a Büchi acceptance condition $\alpha \subseteq Q$ if and only if $S \cap \alpha \neq \emptyset$. Dually, S satisfies a *co-Büchi* acceptance condition $\alpha \subseteq Q$ if and only if $S \cap \alpha = \emptyset$. We say that S is α-free if $S \cap \alpha = \emptyset$. An automaton accepts a word iff it has an accepting run on it. The language of an automaton $\mathcal{A}$, denoted $L(\mathcal{A})$, is the set of words that $\mathcal{A}$ accepts. We also say that $\mathcal{A}$ *recognizes* the language $L(\mathcal{A})$. For two automata $\mathcal{A}$ and $\mathcal{A}'$, we say that $\mathcal{A}$ and $\mathcal{A}'$ are *equivalent* if $L(\mathcal{A}) = L(\mathcal{A}')$.

We denote the different classes of automata by three letter acronyms in $\{\mathrm{D}, \mathrm{N}\} \times \{\mathrm{B}, \mathrm{C}\} \times \{\mathrm{W}\}$. The first letter stands for the branching mode of the automaton (deterministic or nondeterministic); the second letter stands for the acceptance-condition type (Büchi, or co-Büchi); the third letter indicates that the automaton runs on words.

Different classes of automata have different expressive power. In particular, while NBWs recognize all ω-regular language [13], DBWs are strictly less expressive than NBWs, and so are DCWs [12]. In fact, a language L can be recognized by a DBW iff its complement can be recognized by a DCW. Indeed, by viewing a DBW as a DCW, we get an automaton for the complementing language, and vice versa. The expressiveness superiority of the nondeterministic model over the deterministic one does not apply to the co-Büchi acceptance condition. There, every NCW has an equivalent DCW.[2]

3 From NBW to NCW

In this section we describe our main result and point to a family of languages $L_1, L_2, \ldots$ such that for all $k \geq 2$, an NBW for L_k requires strictly fewer states than an NCW for L_k.

[2] When applied to universal Büchi automata, the translation in [14], of alternating Büchi automata into NBW, results in DBW. By dualizing it, one gets a translation of NCW to DCW.

3.1 The Languages L_k

We define an infinite family of languages $L_1, L_2, \ldots$ over the alphabet $\Sigma = \{a, b\}$. For every $k \geq 1$, the language L_k is defined as follows:

$$L_k = \{w \in \Sigma^\omega \mid \text{both } a \text{ and } b \text{ appear at least } k \text{ times in } w\}.$$

Since an automaton recognizing L_k must accept every word in which there are at least k a's and k b's, regardless of how the letters are ordered, it may appear as if the automaton must have two k-counters operating in parallel, which requires $O(k^2)$ states. This would indeed be the case if a and b were not the only letters in Σ, or if the automaton was deterministic. However, since we are interested in nondeterministic automata, and a and b are the only letters in Σ, we can do much better. Since Σ contains only the letters a and b, one of these letters must appear infinitely often in every word in Σ^ω. Hence, $w \in L_k$ iff w has at least k b's and infinitely many a's, or at least k a's and infinitely many b's. An NBW can simply guess which of the two cases above holds, and proceed to validate its guess (if w has infinitely many a's as well as b's, both guesses would succeed). The validation of each of these guesses requires only one k-counter, and a gadget with two states for verifying that there are infinitely many occurrences of the guessed letter. As we later show, implementing this idea results in an NBW with $2k + 1$ states.

Observe that the reason we were able to come up with a very succinct NBW for L_k is that NBW can abstract precise counting by "counting to infinity" with two states. The fact that NCW do not share this ability [12] is what ultimately allows us to prove that NBW are more succinct than NCW. However, it is interesting to note that also NCW for L_k can do much better than $O(k^2)$ states. Even though an NCW cannot validate a guess that a certain letter appears infinitely many times, it does not mean that such a guess is useless. If an NCW guesses that a certain letter appears infinitely many times, then it can postpone counting occurrences of that letter until after it finishes counting k occurrences of the other letter. In other words, $w \in L_k$ iff w has either at least k b's *after* the first k a's, or k a's *after* the first k b's. Following this characterization yields an NCW with two components (corresponding to the two possible guesses) each with two k-counters running sequentially. Since the counters are independent of each other, the resulting NCW has about $4k$ states instead of $O(k^2)$ states. But this is not the end of the story; a more careful look reveals that L_k can also be characterized as follows: $w \in L_k$ iff w has at least k b's *after* the first k a's (this characterizes words in L_k with infinitely many b's), or a finite number of b's that is not smaller than k (this characterizes words in L_k with finitely many b's). Obviously the roles of a and b can also be reversed. As we later show, implementing this idea results in an NCW with $3k + 1$ states. We also show that up to one state this is indeed the best one can do.

3.2 Upper Bounds for L_k

In this section we describe, for every $k \geq 1$, an NBW with $2k + 1$ states and an NCW with $3k + 1$ states that recognize L_k.

Theorem 1. *There is an NBW with $2k + 1$ states that recognizes the language L_k.*

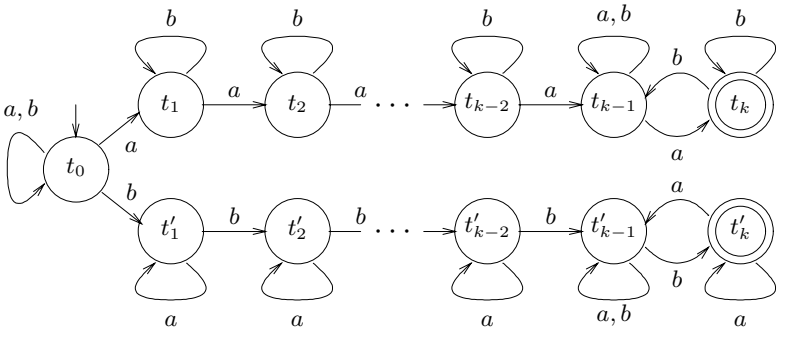


Fig. 1. An NBW for L_k with $2k + 1$ states

Proof: Consider the automaton in Figure 1. Recall that $w \in L_k$ iff w has at least k b's and infinitely many a's, or at least k a's and infinitely many b's. The lower branch of the automaton checks the first option, and the upper branch checks the second option. Let's focus on the upper branch (a symmetric analysis works for the lower branch). The automaton can reach the state marked t_{k-1} iff it can read $k-1$ a's. From the state t_{k-1} the automaton can continue and accept w, iff w has at least one more a (for a total of at least k a's) and infinitely many b's. Note that from t_k the automaton can only read b. Hence, it moves from t_k to t_{k-1} when it guesses that the current b it reads is the last b in a block of consecutive b's (and thus the next letter in the input is a). Similarly, from t_{k-1} the automaton moves to t_k if it reads an a and guesses that it is the last a in a block of consecutive a's. □

Theorem 2. *There is an NCW with* $3k + 1$ *states that recognizes the language* L_k.

Proof: Consider the automaton in Figure 2. Recall that $w \in L_k$ iff w contains at least k b's *after* the first k a's, or a finite number of b's not smaller than k. The upper branch of the automaton checks the first option, and the lower branch checks the second option. It is easy to see that an accepting run using the upper branch first counts k a's, then counts k b's, and finally enters an accepting sink. To see that the lower branch accepts the set of words that have at least k b's, but only finitely many b's, observe that every accepting run using the lower branch proceeds as follows: It stays in the initial state until it guesses that only k b's remain in the input, and then it validates this guess by counting k b's and entering a state from which only a^ω can be accepted. □

Before we turn to study lower bounds for the language L_k, let us note that the strategies used in the NBW and NCW in Figures 1 and 2 are very different. Indeed, the first uses the ability of the Büchi condition to track that an event occurs infinitely often, and the second uses the ability of the co-Büchi condition to track that an event occurs only finitely often. Thus, it is not going to be easy to come up with a general linear translation of NBWs to NCWs that given the NBW in Figure 1 would generate the NCW in Figure 2.

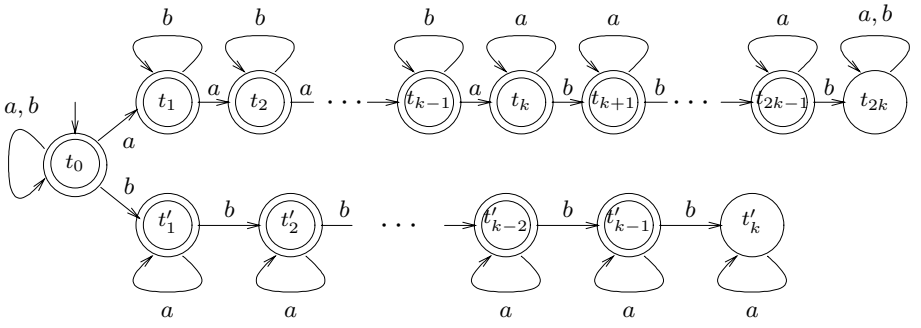


Fig. 2. An NCW for L_k with $3k + 1$ states

3.3 Lower Bounds for L_k

In this section we prove that the constructions in Section 3.2 are optimal. In particular, this section contains our main technical contribution – a lower bound on the number of states of an NCW that recognizes L_k.

Let $\mathcal{A} = \langle \Sigma, Q, \delta, Q_0, \alpha \rangle$ be an NBW or an NCW that recognizes the language L_k. Let $q_0^a q_1^a q_2^a \cdots$ be an accepting run of $\mathcal{A}$ on the word $a^k b^\omega$, and let $q_0^b q_1^b q_2^b \cdots$ be an accepting run of $\mathcal{A}$ on the word $b^k a^\omega$. Also, let $Q_a = \{q_1^a, q_2^a, \ldots, q_k^a\}$, and $Q_b = \{q_1^b, q_2^b, \ldots, q_k^b\}$. Note that $\mathcal{A}$ may have several accepting runs on $a^k b^\omega$ and $b^k a^\omega$, thus there may be several possible choices of Q_a and Q_b. The analysis below is independent of this choice. Observe that for every $1 \leq i \leq k$, the state q_i^a can be reached from Q_0 by reading a^i, and from it the automaton can accept the word $a^{k-i} b^\omega$. Similarly, the state q_i^b can be reached from Q_0 by reading b^i, and from it the automaton can accept the word $b^{k-i} a^\omega$. A consequence of the above observation is the following lemma.

Lemma 1. *The sets Q_a and Q_b are disjoint, of size k each, and do not intersect Q_0.*

Proof: In order to see that $|Q_a| = k$, observe that if $q_i^a = q_j^a$ for some $1 \leq i < j \leq k$, then $\mathcal{A}$ accepts the word $a^i a^{k-j} b^\omega$, which is impossible since it has less than k a's. A symmetric argument shows that $|Q_b| = k$. In order to see that $Q_a \cap Q_b = \emptyset$, note that if $q_i^a = q_j^b$ for some $1 \leq i, j \leq k$, then $\mathcal{A}$ accepts the word $a^i b^{k-j} a^\omega$, which is impossible since it has less than k b's. Finally, if $q_i^a \in Q_0$ for some $1 \leq i \leq k$, then $\mathcal{A}$ accepts the word $a^{k-i} b^\omega$, which is impossible since it has less than k a's. A symmetric argument shows that $Q_b \cap Q_0 = \emptyset$. □

Since obviously $|Q_0| \geq 1$, we have the following.

Theorem 3. *Every NCW or NBW that recognizes L_k has at least $2k + 1$ states.*

Theorem 3 implies that the upper bound in Theorem 1 is tight, thus the case for NBW is closed. In order to show that every NCW $\mathcal{A} = \langle \Sigma, Q, \delta, Q_0, \alpha \rangle$ that recognizes the language L_k has at least $3k$ states, we prove the next two lemmas.

Lemma 2. *If $\mathcal{A} = \langle \Sigma, Q, \delta, Q_0, \alpha \rangle$ is an NCW that recognizes the language L_k, then there are two (not necessarily different) states $q_a, q_b \notin \alpha$, such that q_a and q_b are reachable from each other using α-free paths, and satisfy that $\mathcal{A}$ can accept the word a^ω from q_a, and the word b^ω from q_b.*

Proof: Let n be the number of states in $\mathcal{A}$, and let $r = r_0, r_1, \cdots$ be an accepting run of $\mathcal{A}$ on the word $(a^n b^n)^\omega$. Observe that since r is an accepting run then $inf(r) \cap \alpha = \emptyset$. Since $\mathcal{A}$ has a finite number of states, there exists $l \geq 0$ such that all the states visited after reading $(a^n b^n)^l$ are in $inf(r)$. Furthermore, there must be a state q_a that appears twice among the $n+1$ states $r_{2nl}, \cdots, r_{2nl+n}$ that r visits while reading the $(l+1)-th$ block of a's. It follows that there is $1 \leq m_a \leq n$ such that q_a can be reached from q_a by reading a^{m_a} while going only through states not in α. Similarly, there is a state $q_b \in inf(r)$ and $1 \leq m_b \leq n$ such that q_b can be reached from q_b by reading b^{m_b} while going only through states not in α. Hence, $\mathcal{A}$ can accept the word $(a^{m_a})^\omega = a^\omega$ from q_a, and the word $(b^{m_b})^\omega = b^\omega$ from q_b. Since q_a and q_b appear infinitely often on the α-free tail $r_{2nl}, \cdots$ of r, they are reachable from each other using α-free paths. □

Note that a similar lemma for NBW does not hold. For example, the NBW in Figure 1 is such that there is no state from which a^ω can be accepted, and that can be reached from a state from which b^ω can be accepted. Also note that there may be several possible choices for q_a and q_b, and that our analysis is independent of such a choice.

Lemma 3. *Every simple path from Q_0 to q_a is of length at least $k+1$ and its k-tail is disjoint from $Q_0 \cup Q_a$. Similarly, every simple path from Q_0 to q_b is of length at least $k+1$ and its k-tail is disjoint from $Q_0 \cup Q_b$.*

Proof: We prove the lemma for a path π from Q_0 to q_a (a symmetric argument works for a path to q_b). By Lemma 2, $\mathcal{A}$ can accept the word a^ω from q_a. Hence, $\mathcal{A}$ must read at least k b's before reaching q_a. This not only implies that π is of length at least $k+1$, but also that no state in the k-tail of π can be reached (in zero or more steps) from Q_0 without reading b's. Since all states in $Q_0 \cup Q_a$ violate this requirement, we are done. □

Lemmas 1 and 3 together imply that if there exists a simple path π from Q_0 to q_a whose k-tail is disjoint from Q_b (alternatively, a simple path from Q_0 to q_b whose k-tail is disjoint from Q_a), then $\mathcal{A}$ has at least $3k+1$ states: Q_0, Q_a, Q_b, and the k-tail of π. The NCW used to establish the upper bound in Theorem 2 indeed has such a path. Unfortunately, this is not the case for every NCW recognizing L_k. However, as the next two lemmas show, if the k-tail of π is α-free we can "compensate" for each state (except for q_k^b) common to π and Q_b, which gives us the desired $3k$ lower bound. The proof of the main theorem then proceeds by showing that if we fail to find a simple path from Q_0 to q_a whose k-tail is disjoint from Q_b, and we also fail to find a simple path from Q_0 to q_b whose k-tail is disjoint from Q_a, then we can find a simple path from Q_0 to q_a whose k-tail is α-free.

Lemma 4. *There is a one-to-one function $f_a : Q_a \setminus (\{q_k^a\} \cup \alpha) \to \alpha \setminus (Q_0 \cup Q_a \cup Q_b)$. Similarly, there is a one-to-one function $f_b : Q_b \setminus (\{q_k^b\} \cup \alpha) \to \alpha \setminus (Q_0 \cup Q_a \cup Q_b)$.*

Proof: We prove the lemma for f_b (a symmetric argument works for f_a). Let n be the number of states in $\mathcal{A}$. Consider some $q_i^b \in Q_b \setminus (\{q_k^b\} \cup \alpha)$. In order to define $f_b(q_i^b)$, take an accepting run $r = r_0, r_1, \cdots$ of $\mathcal{A}$ on the word $b^i a^n b^{k-i} a^\omega$. Among the $n+1$ states $r_i, \cdots, r_{i+n}$ that r visits while reading the sub-word a^n there must be two equal states $r_{i+m} = r_{i+m'}$, where $0 \leq m < m' \leq n$. Since the word $b^i a^m (a^{m'-m})^\omega$ has less than k b's it must be rejected. Hence, there has to be a state $s_i \in \alpha$ along the path $r_{i+m}, \cdots, r_{i+m'}$. We define $f_b(q_i^b) = s_i$. Note that s_i can be reached from Q_0 by reading a word with only i b's, and that $\mathcal{A}$ can accept from s_i a word with only $k-i$ b's. We prove that $s_i \notin Q_0 \cup Q_a \cup Q_b$.

- $s_i \notin Q_0 \cup Q_a \cup \{q_1^b, \ldots q_{i-1}^b\}$ because all states in $Q_0 \cup Q_a \cup \{q_1^b, \ldots q_{i-1}^b\}$ can be reached (in zero or more steps) from Q_0 by reading less than i b's, and from s_i the automaton can accept a word with only $k-i$ b's.
- $s_i \neq q_i^b$ since $s_i \in \alpha$ and $q_i^b \notin \alpha$.
- $s_i \notin \{q_{i+1}^b, \ldots q_k^b\}$ because s_i can be reached from Q_0 by reading a word with only i b's, and from all states in $\{q_{i+1}^b, \ldots q_k^b\}$ the automaton can accept a word with less than $k-i$ b's.

It is left to prove that f_b is one-to-one. To see that, observe that if for some $1 \leq i < j \leq k$ we have that $s_i = s_j$, then the automaton would accept a word with only $i + (k-j)$ b's, which is impossible since $i + (k-j) < k$. □

The following lemma formalizes our counting argument.

Lemma 5. *If there is a simple path π from Q_0 to q_a, or from Q_0 to q_b, such that the k-tail of π is α-free, then $\mathcal{A}$ has at least $3k$ states.*

Proof: We prove the lemma for a path π from Q_0 to q_a (a symmetric argument works for a path to q_b). By Lemma 1, it is enough to find $k-1$ states disjoint from $Q_0 \cup Q_a \cup Q_b$. Let $P \subseteq Q_b$ be the subset of states of Q_b that appear on the k-tail of π, and let R be the remaining $k - |P|$ states of this k-tail. By Lemma 3 we have that R is disjoint from $Q_0 \cup Q_a$, and by definition it is disjoint from Q_b. We have thus found $k - |P|$ states disjoint from $Q_0 \cup Q_a \cup Q_b$. It remains to find a set of states S which is disjoint from $Q_0 \cup Q_a \cup Q_b \cup R$, and is of size at least $|P| - 1$. Since the k-tail of π is α-free, it follows from Lemma 4 that for every state q_i^b in P, except maybe q_k^b, there is a "compensating" state $f_b(q_i^b) \in \alpha \setminus (Q_0 \cup Q_a \cup Q_b)$. We define S to be the set $S = \bigcup_{\{q_i^b \in P, q_i^b \neq q_k^b\}} \{f_b(q_i^b)\}$ of all these compensating states. Since f_b is one-to-one S is of size at least $|P| - 1$. Since R is α-free and $S \subseteq \alpha$ it must be that S is also disjoint from R, and we are done. □

We are now ready to prove our main theorem.

Theorem 4. *Every NCW that recognizes the language L_k has at least $3k$ states.*

Proof: As noted earlier, by Lemmas 1 and 3, if there exists a simple path from Q_0 to q_a whose k-tail is disjoint from Q_b, or if there exists a simple path from Q_0 to q_b whose k-tail is disjoint from Q_a, then $\mathcal{A}$ has at least $3k+1$ states: Q_0, Q_a, Q_b, and the k-tail of this path. We thus assume that on the k-tail of every simple path from Q_0 to q_a there

is a state from Q_b, and that on the k-tail of every simple path from Q_0 to q_b there is a state from Q_a. Note that since by Lemma 3 the k-tail of every simple path from Q_0 to q_b is disjoint from Q_b, it follows from our assumption that $q_a \neq q_b$.

Another consequence of our assumption is that q_a is reachable from Q_b. Take an arbitrary simple path from Q_b to q_a, let q_i^b be the last state in Q_b on this path, and let $q_i^b = v_0, \cdots, v_h = q_a$ be the tail of this path starting at q_i^b. Note that if $q_a \in Q_b$ then $h = 0$. Define π^a to be the path $q_0^b, \cdots, q_i^b, v_1, \cdots, v_h$. Observe that by Lemma 1, and the fact that $v_1, ..., v_h$ are not in Q_b, the path π^a is simple. Hence, by our assumption, the k-tail of π^a intersects Q_b. Since $v_1, ..., v_h$ are not in Q_b, it must be that $h < k$.

By Lemma 2, q_b is reachable from q_a without using states in α. Thus, there exists a simple α-free path $q_a = u_0, ..., u_m = q_b$. Since $u_0 = q_a \in \pi^a$, we can take $0 \leq j \leq m$ to be the maximal index such that u_j appears on π^a. Define the path π^b, from Q_0 to q_b, to be the prefix of π^a until (but not including) u_j, followed by the path $u_j, ..., u_m$. Note that π^b is a simple path since by our choice of u_j it is the concatenation of two disjoint simple paths. Hence, by our assumption, there is some state $q_j^a \in Q_a$ on the k-tail of π^b. We claim that q_j^a must be on the α-free tail $u_j, ..., u_m$ of π^b. Recall that all the states in π^b before u_j are also in π^a, so it is enough to prove that q_j^a is not in π^a. By Lemma 1, q_j^a cannot be equal to any of the first $i+1$ states of π^a. By Lemma 3, and the fact that $h < k$, it cannot be equal to any of the remaining h states of π^a. We can thus conclude that the tail of π^b starting at q_j^a is α-free.

We are now in a position to build a new simple path π from Q_0 to q_a, whose k-tail is α-free. By Lemma 5, this completes the proof. We first define a path π' from Q_0 to q_a by concatenating to the path $q_0^a, q_1^a, \cdots q_{j-1}^a$ the tail of π^b starting at q_j^a, followed by some α-free path from q_b to q_a (by Lemma 2 such a path exists). Since π' may have repeating states, we derive from it the required simple path π by eliminating repetitions in an arbitrary way. Observe that the only states in π' (and thus also in π) that may be in α are the states $\{q_0^a, q_1^a, \ldots q_{j-1}^a\}$. By Lemma 3, the k-tail of π is disjoint from $Q_0 \cup Q_a$. Hence, it must be α-free. □

Combining the upper bound in Theorem 1 with the lower bound in Theorem 4, we get the following corollary.

Corollary 1. *For every integer $k \geq 1$, there is a language L_k over a two-letter alphabet, such that L_k can be recognized by an NBW with $2k+1$ states, whereas the minimal NCW that recognizes L_k has $3k$ states.*

4 From NCW to NBW

As shown in Section 3, NBWs are more succinct than NCWs. In this section we study the translation of NCW to NBW and show that the converse is also true. That is, we show that the known construction that translates an NCW with n states and acceptance condition α, to an equivalent NBW with $2n - |\alpha|$ states, is tight. For reference, we first briefly recall this translation. The translation we present follows [10], which complements deterministic Büchi automata.

Theorem 5. *[10] Given an NCW $\mathcal{A} = \langle \Sigma, Q, \delta, Q_0, \alpha \rangle$ with n states, one can build an equivalent NBW $\mathcal{A}'$ with $2n - |\alpha|$ states.*

Proof: The NBW $\mathcal{A}'$ is built by taking two copies of $\mathcal{A}$, deleting all the states in α from the second copy, and making all the remaining states of the second copy accepting. Transitions are also added to enable the automaton to move from the first copy to the second copy, but not back. The idea is that since an accepting run of $\mathcal{A}$ visits states in α only finitely many times, it can be simulated by a run of $\mathcal{A}'$ that switches to the second copy when states in α are no longer needed. More formally, $\mathcal{A}' = \langle \Sigma, (Q \times \{0\}) \cup ((Q \setminus \alpha) \times \{1\}), \delta', Q_0 \times \{0\}, (Q \setminus \alpha) \times \{1\} \rangle$, where for every $q \in Q$ and $\sigma \in \Sigma$ we have $\delta'(\langle q, 0 \rangle, \sigma) = (\delta(q, \sigma) \times \{0\}) \cup ((\delta(q, \sigma) \setminus \alpha) \times \{1\}$, and for every $q \in Q \setminus \alpha$ and $\sigma \in \Sigma$ we have $\delta'(\langle q, 1 \rangle, \sigma) = (\delta(q, \sigma) \setminus \alpha) \times \{1\}$. □

Observe that if $\alpha = \emptyset$, then $L(\mathcal{A}) = \Sigma^*$, and the translation is trivial. Hence, the maximal possible blowup is when $|\alpha| = 1$. In the remainder of this section we prove that there are NCWs (in fact, DCWs with $|\alpha| = 1$) for which the $2n - |\alpha|$ blowup cannot be avoided.

4.1 The Languages L'_k

We define a family of languages $L'_2, L'_3, \ldots$ over the alphabet $\Sigma = \{a, b\}$. For every $k \geq 2$ we let $L'_k = (a^k b^k + a^k b^{k-1})^* (a^k b^{k-1})^\omega$. Thus, a word $w \in \{a, b\}^\omega$ is in L'_k iff w begins with an a, all the blocks of consecutive a's in w are of length k, all the blocks of consecutive b's in w are of length k or $k-1$, and only finitely many blocks of consecutive b's in w are of length k. Intuitively, an automaton for L'_k must be able to count finitely many times up to $2k$, and infinitely many times up to $2k-1$. The key point is that while a co-Büchi automaton can share the states of the $2k-1$ counter with those of the $2k$ counter, a Büchi automaton cannot.

4.2 Upper Bounds for L'_k

We first describe an NCW (in fact, a DCW) with $2k$ states that recognizes the language L'_k. By Theorem 5, one can derive from it an equivalent NBW with $4k-1$ states.

Theorem 6. *There is a DCW with $2k$ states that recognizes the language L'_k.*

Proof: Consider the automaton in Figure 3. It is obviously deterministic, and it is easy to see that it accepts the language $a^k b^{k-1} (a^k b^{k-1} + b a^k b^{k-1})^* (a^k b^{k-1})^\omega = (a^k b^k + a^k b^{k-1})^* (a^k b^{k-1})^\omega = L'_k$. □

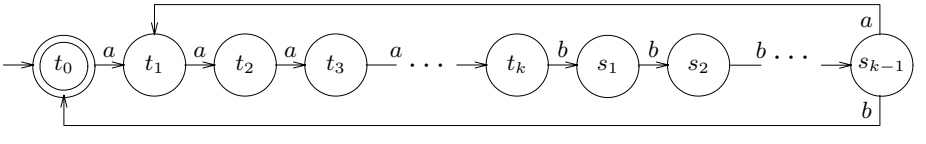


Fig. 3. A DCW for L'_k with $2k$ states

Note that the NCW in Figure 3 is really a DCW, thus the lower bound we are going to prove is for the DCW to NBW translation. It is worth noting that the dual translation, of DBW to NCW (when exists), involves no blowup. Indeed, if a DBW $\mathcal{A}$ recognizes a language that is recognizable by an NCW, then this language is also recognizable by a DCW, and there is a DCW on the same structure as $\mathcal{A}$ for it [6,8].

4.3 Lower Bounds for L'_k

In this section we prove that the NBW obtained by applying the construction in Theorem 5 to the automaton in Figure 3, is optimal. Thus, every NBW for L'_k has at least $4k-1$ states. Note that this also implies that the upper bound in Theorem 6 is tight too.

We first show that an automaton for L'_k must have a cycle along which it can count to $2k$, for the purpose of keeping track of occurrences of $a^k b^k$ in the input.

Lemma 6. *If $\mathcal{A} = \langle \Sigma, Q, \delta, Q_0, \alpha \rangle$ is an NCW or an NBW that recognizes the language L'_k, then there is a cycle C, reachable from Q_0, with at least $2k$ different states, along which $\mathcal{A}$ can traverse a finite word containing the substring $a^k b^k$.*

Proof: Let n be the number of states in $\mathcal{A}$, and let $r = r_0, r_1, \cdots$ be an accepting run of $\mathcal{A}$ on the word $w = (a^k b^k)^{n+1}(a^k b^{k-1})^\omega$. Since $\mathcal{A}$ has only n states, there must be $1 \leq i < j \leq n+1$, such that $r_{i2k} = r_{j2k}$. Consider the cycle $C = r_{i2k}, \cdots, r_{j2k-1}$. Note that $r_0 \in Q_0$ and thus C is reachable from Q_0. Also note that $j - i \geq 1$, and that $\mathcal{A}$ can traverse $(a^k b^k)^{j-i}$ along C.

We now prove that the states $r_{i2k}, \cdots, r_{(i+1)2k-1}$ are all different, thus C has at least $2k$ different states. Assume by way of contradiction that this is not the case, and let $0 \leq h < l \leq 2k-1$ be such that $r_{i2k+h} = r_{i2k+l}$. Define $u = a^k b^k$, and let $u = xyz$, where $x = u_1 \cdots u_h$, $y = u_{h+1} \cdots u_l$, and $z = u_{l+1} \cdots u_{2k}$. Observe that x and z may be empty, and that since $0 \leq h < l \leq 2k-1$, it must be that $0 < |y| < 2k$. Also note that $\mathcal{A}$ can traverse x along $r_{i2k} \cdots r_{i2k+h}$, and traverse y along the cycle $\hat{C} = r_{i2k+h}, \cdots, r_{i2k+l-1}$. By adding k more traversals of the cycle $\hat{C}$ we can derive from r a run $r' = r_0 \cdots r_{i2k+h} \cdot (r_{i2k+h+1} \cdots r_{i2k+l})^{k+1} \cdot r_{i2k+l+1} \cdots$ on the word $w' = (a^k b^k)^i x y^{k+1} z (a^k b^k)^{n-i} (a^k b^{k-1})^\omega$. Similarly, by removing from r a traversal of $\hat{C}$, we can derive a run $r'' = r_0 \cdots r_{i2k+h} r_{i2k+l+1} \cdots$ on the word $w'' = (a^k b^k)^i x z (a^k b^k)^{n-i} (a^k b^{k-1})^\omega$. Since $inf(r) = inf(r') = inf(r'')$, and r is accepting, so are r' and r''. Hence, w' and w'' are accepted by $\mathcal{A}$.

To derive a contradiction, we show that $w' \notin L'_k$ or $w'' \notin L'_k$. Recall that $xyz = a^k b^k$ and that $0 < |y| < 2k$. Hence, there are two cases to consider: either $y \in a^+ + b^+$, or $y \in a^+ b^+$. In the first case we get that y^{k+1} contains either a^{k+1} or b^{k+1}, which implies that $w' \notin L'_k$. Consider now the case $y \in a^+ b^+$. Let $y = a^m b^t$. Since $i > 0$, the prefix $(a^k b^k)^i x z a^k$ of w'' ends with $b^k a^{k-m} b^{k-t} a^k$. Since all the consecutive blocks of a's in w must be of length k, and $m > 0$, it must be that $k - m = 0$. Hence, w'' contains the substring $b^k b^{k-t}$. Recall that $k = m$, and that $m + t < 2k$. Thus, $k - t > 0$, and $b^k b^{k-t}$ is a string of more than k consecutive b's. Since no word in L'_k contains such a substring, we are done. □

The following lemma shows that an NBW recognizing L'_k must have a cycle going through an accepting state along which it can count to $2k-1$, for the purpose of recognizing the $(a^k b^{k-1})^\omega$ tail of words in L'_k.

Lemma 7. *If $\mathcal{A} = \langle \Sigma, Q, \delta, Q_0, \alpha \rangle$ is an NBW that recognizes the language L'_k, then $\mathcal{A}$ has a cycle C, with at least $2k-1$ different states, such that $C \cap \alpha \neq \emptyset$.*

Proof: Since $L(\mathcal{A})$ is not empty, there must be a state $c_0 \in \alpha$ that is reachable from Q_0, and a simple cycle $C = c_0, \cdots, c_{m-1}$ going through c_0. Since C is simple, all its states are different. It remains to show that $m \geq 2k-1$. Let $u \in \Sigma^*$ be such that $\mathcal{A}$ can reach c_0 from Q_0 while reading u, and let $v = \sigma_1 \cdots \sigma_m$ be such that $\mathcal{A}$ can traverse v along C. It follows that $w = uv^\omega$ is accepted by $\mathcal{A}$. Since all words in L'_k have infinitely many a's and b's, it follows that a and b both appear in v. We can thus let $1 \leq j < m$ be such that $\sigma_j \neq \sigma_{j+1}$. Let x be the substring $x = \sigma_j \cdots \sigma_m \sigma_1 \cdots \sigma_{j+1}$ of vv. Since $\sigma_j \neq \sigma_{j+1}$, it must be that x contains one block of consecutive letters all equal to σ_{j+1} that *starts* at the second letter of x, and another block of consecutive letters all equal to σ_j that *ends* at the letter before last of x. Since $|x| = m+2$ we have that x contains at least one block of consecutive a's and one block of consecutive b's that start and end within the span of m letters. Recall that since $w \in L'_k$ then all the blocks of consecutive a's in v^ω must be of length k, and all the blocks of consecutive b's in v^ω must be of length at least $k-1$. Hence, $m \geq k+k-1$. □

Theorem 7. *Every NBW $\mathcal{A} = \langle \Sigma, Q, \delta, Q_0, \alpha \rangle$ that recognizes the language L'_k has at least $4k-1$ states.*

Proof: By Lemma 6, there is a cycle $C = c_0, \cdots, c_{n-1}$, reachable from Q_0, with at least $2k$ different states, along which $\mathcal{A}$ can read some word $z = z_1 \cdots z_n$ containing $a^k b^k$. By Lemma 7, there is a cycle $C' = c'_0, \cdots, c'_{m-1}$, with at least $2k-1$ different states, going through an accepting state $c'_0 \in \alpha$. In order to prove that $\mathcal{A}$ has at least $4k-1$ states, we show that C and C' are disjoint. Assume by way of contradiction that there is a state $q \in C \cap C'$. By using q as a pivot we can construct a run r that alternates infinitely many times between the cycles C and C'. Since C' contains an accepting state, the run r is accepting. To reach a contradiction, we show that r is a run on a word containing infinitely many occurrences of b^k, and thus it must be rejecting. Let $0 \leq l < n$ and $0 \leq h < m$ be such that $q = c_l = c'_h$, and let $q_0, \cdots, q_t$ be a path from Q_0 to q (recall that C is reachable from Q_0). Consider the run $r = q_0 \cdots q_{t-1}(c'_h \cdots c'_{m-1} c'_0 \cdots c'_{h-1} c_l \cdots c_{n-1} c_0 \cdots c_{n-1} c_0 \cdots c_{l-1})^\omega$. Let $x, y \in \Sigma^*$ be such that $\mathcal{A}$ can read x along the path $q_0, \cdots q_t$, and read y while going from c'_h back to itself along the cycle C'. Observe that r is a run of $\mathcal{A}$ on the word $w = x \cdot (y \cdot z_{l+1} \cdots z_n \cdot z \cdot z_1 \cdots z_l)^\omega$. Since $c'_0 \in \alpha$ and r goes through c'_0 infinitely many times, r is an accepting run of $\mathcal{A}$ on w. Since w contains infinitely many occurrences of z it contains infinitely many occurrences of b^k, and thus $w \notin L'_k$, which is a contradiction. □

Combining the upper bound in Theorem 6 with the lower bound in Theorem 7 we get the following corollary:

Corollary 2. *For every integer $k \geq 2$, there is a language L'_k over a two-letter alphabet, such that L'_k can be recognized by a DCW with $2k$ states, whereas the minimal NBW that recognizes L'_k has $4k-1$ states.*

5 Discussion

We have shown that NBWs are more succinct than NCWs. The advantage of NBWs that we used is their ability to save states by counting to infinity with two states instead of counting to k, for some parameter k. The bigger k is, the bigger is the saving. In our lower bound proof, k is linear in the size of the state space. Increasing k to be exponential in the size of the state space would lead to an exponential lower bound for the NBW to NCW translation. Once we realized this advantage of the Büchi condition, we tried to find an NBW that uses a network of nested counters in a way that would enable us to increase the relative size of k. We did not find such an NBW, and we conjecture that the succinctness of the Büchi condition cannot go beyond saving one copy of the state space. Let us elaborate on this.

The best known upper bound for the NBW to NCW translation is still exponential, and the upper bound for the NCW to NBW translation is linear. Still, it was much easier to prove the succinctness of NCWs with respect to NBWs (Section 4) than the succinctness of NBWs with respect to NCWs (Section 3). Likewise, DCWs are more succinct than NBWs (Section 4), whereas DBWs are not more succinct than NCWs [6]. The explanation for this quite counterintuitive "ease of succinctness" of the co-Büchi condition is the expressiveness superiority of the Büchi condition. Since every NCW has an equivalent NBW, all NCWs are candidates for proving the succinctness of NCW. On the other hand, only NBWs that have an equivalent NCW are candidates for proving the succinctness of NBWs. Thus, the candidates have to take an advantage of the strength of the Büchi condition, but at the same time be restricted to the co-Büchi condition. This restriction has caused researchers to believe that NBWs are actually co-Büchi-type (that is, if an NBW has an equivalent NCW, then it also has an equivalent NCW on the same structure). The results in [8] refuted this hope, and our results here show that NBWs can actually use their expressiveness superiority for succinctness. While the results are the first to show such a succinctness, our fruitless efforts to improve the lower bound further have led us to believe that NBWs cannot do much more than abstracting counting up to the size of the state space. Intuitively, as soon as the abstracted counting goes beyond the size of the state space, the language has a real "infinitely often" nature, and it is not recognizable by an NCW. Therefore, our future research focuses on improving the upper bound. The very different structure and strategy behind the NBW and NCW in Figures 1 and 2 hint that this is not going to be an easy journey either.

References

1. Armoni, R., Fix, L., Flaisher, A., Gerth, R., Ginsburg, B., Kanza, T., Landver, A., Mador-Haim, S., Singerman, E., Tiemeyer, A., Vardi, M.Y., Zbar, Y.: The ForSpec temporal logic: A new temporal property-specification logic. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 196–211. Springer, Heidelberg (2002)
2. Beer, I., Ben-David, S., Eisner, C., Fisman, D., Gringauze, A., Rodeh, Y.: The temporal logic Sugar. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 363–367. Springer, Heidelberg (2001)
3. Büchi, J.R.: On a decision method in restricted second order arithmetic. In: Proc. Int. Congress on Logic, Method, and Philosophy of Science, 1960, pp. 1–12. Stanford University Press (1962)

4. Emerson, E.A., Jutla, C.: The complexity of tree automata and logics of programs. In: Proc. 29th IEEE Symp. on Foundations of Computer Science, pp. 328–337 (1988)
5. Accellera Organization Inc (2006), http://www.accellera.org
6. Krishnan, S.C., Puri, A., Brayton, R.K.: Deterministic ω-automata vis-a-vis deterministic Büchi automata. In: Du, D.-Z., Zhang, X.-S. (eds.) ISAAC 1994. LNCS, vol. 834, pp. 378–386. Springer, Heidelberg (1994)
7. Kupferman, O.: Tightening the exchange rate beteen automata. In: Duparc, J., Henzinger, T.A. (eds.) CSL 2007. LNCS, vol. 4646, pp. 7–22. Springer, Heidelberg (2007)
8. Kupferman, O., Morgenstern, G., Murano, A.: Typeness for ω-regular automata. In: Wang, F. (ed.) ATVA 2004. LNCS, vol. 3299, pp. 324–338. Springer, Heidelberg (2004)
9. Kupferman, O., Vardi, M.Y.: From linear time to branching time. ACM Transactions on Computational Logic 6(2), 273–294 (2005)
10. Kurshan, R.P.: Complementing deterministic Büchi automata in polynomial time. Journal of Computer and Systems Science 35, 59–71 (1987)
11. Kurshan, R.P.: Computer Aided Verification of Coordinating Processes. Princeton Univ. Press, Princeton (1994)
12. Landweber, L.H.: Decision problems for ω–automata. Mathematical Systems Theory 3, 376–384 (1969)
13. McNaughton, R.: Testing and generating infinite sequences by a finite automaton. Information and Control 9, 521–530 (1966)
14. Miyano, S., Hayashi, T.: Alternating finite automata on ω-words. Theoretical Computer Science 32, 321–330 (1984)
15. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Proc. 16th ACM Symp. on Principles of Programming Languages, pp. 179–190 (1989)
16. Rabin, M.O.: Decidability of second order theories and automata on infinite trees. Transaction of the AMS 141, 1–35 (1969)
17. Ravi, K., Bloem, R., Somenzi, F.: A comparative study of symbolic algorithms for the computation of fair cycles. In: Johnson, S.D., Hunt Jr., W.A. (eds.) FMCAD 2000. LNCS, vol. 1954, pp. 143–160. Springer, Heidelberg (2000)
18. Safra, S.: On the complexity of ω-automata. In: Proc. 29th IEEE Symp. on Foundations of Computer Science, pp. 319–327 (1988)
19. Street, R.S., Emerson, E.A.: An elementary decision procedure for the μ-calculus. In: Paredaens, J. (ed.) ICALP 1984. LNCS, vol. 172, pp. 465–472. Springer, Heidelberg (1984)
20. Vardi, M.Y., Wolper, P.: Automata-theoretic techniques for modal logics of programs. Journal of Computer and Systems Science 32(2), 182–221 (1986)
21. Vardi, M.Y., Wolper, P.: Reasoning about infinite computations. Information and Computation 115(1), 1–37 (1994)
22. Wilke, T.: CTL^+ is exponentially more succinct than CTL. In: Pandu Rangan, C., Raman, V., Ramanujam, R. (eds.) FST TCS 1999. LNCS, vol. 1738, pp. 110–121. Springer, Heidelberg (1999)
23. Wolper, P., Vardi, M.Y., Sistla, A.P.: Reasoning about infinite computation paths. In: Proc. 24th IEEE Symp. on Foundations of Computer Science, pp. 185–194 (1983)

Recurrent Reachability Analysis in Regular Model Checking

Anthony Widjaja To and Leonid Libkin

LFCS, School of Informatics, University of Edinburgh
{anthony.w.to,libkin}@ed.ac.uk

Abstract. We consider the problem of recurrent reachability over infinite systems given by regular relations on words and trees, i.e, whether a given regular set of states can be reached infinitely often from a given initial state in the given transition system. Under the condition that the transitive closure of the transition relation is regular, we show that the problem is decidable, and the set of all initial states satisfying the property is regular. Moreover, our algorithm constructs an automaton for this set in polynomial time, assuming that a transducer of the transitive closure can be computed in poly-time. We then demonstrate that transition systems generated by pushdown systems, regular ground tree rewrite systems, and the well-known process algebra PA satisfy our condition and transducers for their transitive closures can be computed in poly-time. Our result also implies that model checking EF-logic extended by recurrent reachability predicate (EGF) over such systems is decidable.

1 Introduction

Infinite-state systems play an important role in verification as they capture many scenarios that cannot be adequately described by standard finite-state models. For example, the behavior of parameterized systems needs to be checked regardless of the number of processes, and this is often most suitably represented by an infinite-state system.

The most common verification problems for such systems can be abstracted as reachability and recurrent reachability [2, 3, 8, 9]. Reachability asks if a given state, or a state in a given set, can be reached from an initial state. Checking these is essential for verifying safety of infinite-state systems, as we want to find counterexamples to specifications saying that bad states cannot be reached. If we have slightly weaker specifications saying that undesirable states can only appear in some initial portion of each execution path, then counterexamples to those are formalized as *recurrent reachability*, i.e. the existence of a witnessing path that infinitely often goes through a given set of states. In the CTL* notation, recurrent reachability for a set L is $\mathbf{EGF}L$. Observe that although for finite systems recurrent reachability is reducible to reachability, this is not the case for infinite systems in general, e.g. lossy channel systems (see [1]).

To make the questions of model checking meaningful for infinite systems, they need to have an effective finite representation. Often, the state space is described

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 198–213, 2008.

by regular word or tree languages, and transitions are given by regular word or tree transducers: this is the general framework of regular model-checking [2, 3]. Without restrictions, this does not guarantee decidability even for the simplest reachability properties. Hence, one normally restricts the class of transducers so that their iterations would remain regular [7, 17]. Then such infinite-state systems have an effective word-automatic or tree-automatic presentation [5, 6]. Some of the most well-known and most studied classes of such systems include pushdown systems [14, 15, 22, 27], prefix-recognizable graphs [11, 24], ground tree rewrite systems [13, 19], and the process algebra PA [4, 20, 22].

For such systems, reachability has been extensively studied [3, 9, 14, 15, 16, 17, 23, 27]. Much less is known about recurrent reachability. Unlike reachability, it is not immediately seen to be decidable even under the assumption that the transitive closure of the transition relation is representable by a regular transducer. One recent result [18, 19] showed that recurrent reachability is decidable for infinite-state transition systems generated by ground tree rewrite systems.

Our main contributions are as follows. We look at arbitrary infinite-state transition systems that have an automatic representation (either word-automatic or tree-automatic) and that further satisfy the condition that the transitive closure of its transition relation is regular. We then show the following:

1. For every regular language L, the set of all states that satisfy a recurrent reachability property $\mathbf{EGF}L$ is also regular. This observation gives rise to two flavors of the model-checking problem: the global problem is to construct an automaton accepting the set of states satisfying $\mathbf{EGF}L$, and the local problem is to verify whether a given word/tree satisfies the property.
2. We give a *generic* poly-time algorithm that solves both model-checking problems for $\mathbf{EGF}L$ given the following as inputs: word/tree regular transducers defining a transition relation and its transitive closure, and a nondeterministic word/tree automaton defining L. For positive answers, our algorithm also constructs some witnessing infinite paths using Büchi word/tree automata as finite representations. In particular, if the transducer defining the transitive closure can itself be computed in poly-time, we obtain a poly-time algorithm for checking recurrent reachability properties. One can also combine our algorithm with the semi-algorithms for computing iterating transducers developed in regular model checking (e.g. [2, 3, 7, 17]).
3. We then look at some particular examples of transition systems in which the transitive closure of the transition relation is regular for which an iterating transducer is poly-time computable. As corollaries, we obtain poly-time algorithms for recurrent reachability over pushdown systems, ground tree rewrite systems, and PA-processes. These also imply that the extension of the **EF**-logic [8, 19, 23] with the **EGF** operator remains decidable for all those examples. For the first two examples, our results follow from known results [15, 18, 19] proven using specialized methods for pushdown systems and ground tree rewrite systems, although their methods do not show how to compute witnessing paths, which are also of interests in verification. Our results for PA-processes are new.

Outline of the paper In Section 2, we recall some basic definitions. In Section 3 we prove our results for transition systems that have word-automatic presentations, and provide applications to pushdown systems. In Section 4 we prove results for tree-automatic presentations, and provide applications to ground tree rewrite systems and PA-processes. We conclude in Section 5 with future work.

2 Preliminaries

Transition systems. Let $\mathrm{AP} = \{P_1, \ldots, P_n\}$ be a finite set of *atomic propositions*. A *transition system* over AP is

$$\mathcal{S} \;=\; \langle S,\ \rightarrow,\ \lambda \rangle,$$

where S is a set of *states*, $\rightarrow \subseteq S \times S$ is a *transition relation*, and $\lambda : \mathrm{AP} \rightarrow 2^S$ is a function defining which states satisfy any particular atomic proposition. The set S is not required to be finite.

We write $\rightarrow^+$ (resp. $\rightarrow^*$) to denote the transitive (resp. transitive-reflexive) closure of $\rightarrow$. If $S' \subseteq S$, then $pre^*(S')$ (resp. $post^*(S')$) denotes the set of states s that can reach (resp. be reached from) some state in S'.

Recurrent reachability. Given a transition system $\mathcal{S} = \langle S, \rightarrow, \lambda \rangle$ and a set $S' \subseteq S$, we write $s \rightarrow^\omega S'$ iff there exists an infinite sequence $\{s_i\}_{i \in \mathbb{N}}$ such that $s_0 = s$ and $s_i \in S'$ and $s_{i-1} \rightarrow^+ s_i$ for all $i > 0$. By transitivity of $\rightarrow^+$, every infinite subsequence of such a sequence $\{s_i\}_{i \in \mathbb{N}}$ that starts with s_0 is also a witness for $s \rightarrow^\omega S'$. We write $Rec(S')$ to denote the set of states s such that $s \rightarrow^\omega S'$. We will also write $Rec(S', \rightarrow^+)$ to emphasize the transitive binary relation in use.

Words, Trees, and Automata. We assume basic familiarity with automata on finite and infinite words and trees (see [12, 25]). Fix a finite alphabet Σ. For each finite word $w = w_1 \ldots w_n \in \Sigma^*$, we write $w[i, j]$, where $1 \leq i \leq j \leq n$, to denote the segment $w_i \ldots w_j$. Given an automaton $\mathcal{A} = (\Sigma, Q, \delta, q_0, F)$, a run of $\mathcal{A}$ on w is a function $\rho : \{0, \ldots, n\} \rightarrow Q$ with $\rho(0) = q_0$ that obeys δ. In this case, the length $\|\rho\|$ of ρ is n. The last state $\rho(n)$ appearing in ρ is denoted by $\mathbf{last}(\rho)$; the first state $\rho(0)$ is denoted by $\mathbf{first}(\rho)$. We also define *run segments* to be runs that do not necessarily start from q_0. Given a run segment $\rho' : \{0, \ldots, m\} \rightarrow Q$ such that $\mathbf{first}(\rho') = \mathbf{last}(\rho)$, we may concatenate ρ and ρ' to obtain a new run $\rho \odot \rho' : \{0, \ldots, n + m\} \rightarrow Q$ defined in the obvious way. We also use the notation $\rho[i, j]$ to denote the segment $\rho(i) \ldots \rho(j)$. A run on an ω-word $w \in \Sigma^\omega$ is a function $\rho : \mathbb{N} \rightarrow Q$ with $\rho(0) = q_0$ that obeys δ. We use abbreviations NWA and NBWA for nondeterministic (Büchi) word automata.

Given a finite direction alphabet Υ, a *tree domain* is a non-empty prefix-closed set $D \subseteq \Upsilon^*$. The empty word (denoted by ε) is referred to as the root. Words $u \in D$ so that no ui is in D are called *leaves*. A *tree* T is a pair (D, τ), where D is a tree domain and τ is a node-labeling function mapping D to Σ. The tree T is said to be *finite* if D is finite; otherwise, it is said to be *infinite*. The tree T is

said to be *complete* if, whenever $u \in D$, if $ui \in D$ for some $i \in \Upsilon$, then $uj \in D$ for all $j \in \Upsilon$. If T is infinite, it is said to be *full* if $D = \Upsilon^*$. The set of all finite trees over Υ and Σ is denoted by $\textsc{Tree}_\Upsilon(\Sigma)$. If $\Upsilon = \{1, 2\}$, we write $\textsc{Tree}_2(\Sigma)$ for $\textsc{Tree}_\Upsilon(\Sigma)$.

A (top-down) *tree automaton* $\mathcal{A} = (\Sigma, Q, \delta, q_0, F)$ over finite Σ-labeled trees has a transition function $\delta : Q \times \Sigma \to 2^{Q^m}$, where $m = |\Upsilon|$. For our constructions, it will be most convenient to define runs on trees with virtual leaf nodes. We define $\mathbf{virt}(T)$ to be the $(\Sigma \cup \bot)$-labeled Υ-tree (D', τ') such that $D' := D \cup \{vi : v \in D, i \in \Upsilon\}$ and if $u \in D$, then $\tau'(u) := \tau(u)$; if $u \notin D$, then $\tau'(u) = \bot$. Notice that $\mathbf{virt}(T)$ is complete. A *run* of $\mathcal{A}$ on T, i.e. a mapping $\rho : D' \to Q$, starts in the initial state q_0 and for each node u labeled a with children $u1, \ldots, um$, we have $(\rho(u1), \ldots, \rho(um)) \in \delta(q, a)$. A run is *accepting* if $\rho(u) \in F$ for each leaf $u \in D'$.

We abbreviate nondeterministic tree automata as NTA, and write NBTA for tree automata over full infinite trees with a Büchi acceptance condition (that will be sufficient for our purposes). For all kinds of automata, $L(\mathcal{A})$ stands for the word or tree languages accepted by $\mathcal{A}$. Also for all types of automata $\mathcal{A}$, we write $\mathcal{A}^q$ for $\mathcal{A}$ in which the initial state is set to q.

Transducers. These will be given by *letter-to-letter automata* that accept binary (and, more generally, k-ary) relations over words and trees (cf. [5, 6]). We start with words. Given two words $w = w_1 \ldots w_n$ and $w' = w'_1 \ldots w'_m$ over the alphabet Σ, we define a word $w \otimes w'$ of length $k = \max\{n, m\}$ over alphabet $\Sigma_\bot \times \Sigma_\bot$, where $\Sigma_\bot = \Sigma \cup \{\bot\}$ and $\bot \notin \Sigma$, as follows:

$$w \otimes w' \;=\; \begin{bmatrix} a_1 \\ b_1 \end{bmatrix} \ldots \begin{bmatrix} a_k \\ b_k \end{bmatrix}, \text{ where } a_i \;=\; \begin{cases} w_i & i \le n \\ \bot & i > n, \end{cases} \text{ and } b_i \;=\; \begin{cases} w'_i & i \le m \\ \bot & i > m. \end{cases}$$

In other words, the shorter word is padded with $\bot$'s, and the ith letter of $w \otimes w'$ is then the pair of the ith letters of padded w and w'. A letter-to-letter automaton is simply an automaton over $\Sigma_\bot \times \Sigma_\bot$, and a binary relation R over Σ^* is *regular* if the set $\{w \otimes w' : (w, w') \in R\}$ is accepted by a letter-to-letter automaton $\mathcal{R}$. We shall refer to such an automaton as a *transducer* over Σ^*, since it can be alternatively viewed as mapping words $w \in \Sigma^*$ nondeterministically into words w' so that $w \otimes w'$ is accepted by $\mathcal{R}$.

Given two trees $T_1 = (D_1, \tau_1)$ and $T_2 = (D_2, \tau_2)$, we define $T = T_1 \otimes T_2$ as a tree over the labeling alphabet $\Sigma^2_\bot$ similarly to the definition of $w \otimes w'$. That is, the domain of T is $D_1 \cup D_2$, and the labeling $\tau : D_1 \cup D_2 \to \Sigma^2_\bot$ is defined as $\tau(u) = (a_1, a_2)$ so that $a_i = \tau_i(u)$ if $u \in D_i$ and $\bot$ otherwise, for $i = 1, 2$.

As for words, a binary relation R over $\textsc{Tree}_\Upsilon(\Sigma)$ is regular if there is a tree automaton $\mathcal{R}$ over $\textsc{Tree}_\Upsilon(\Sigma^2_\bot)$ accepting the set $\{T_1 \otimes T_2 \mid (T_1, T_2) \in R\}$. We also view it as a transducer that nondeterministically assigns to a tree T_1 any tree T_2 so that $(T_1, T_2) \in R$. If the binary relation R defined by a transducer $\mathcal{R}$ is transitive, we shall refer to $\mathcal{R}$ itself as being transitive.

Automatic transition systems. In this paper, we deal with infinite transition systems that can be finitely represented by word or tree automata. We say that

a transition system $\mathcal{S} = \langle S, \rightarrow, \lambda \rangle$ over AP is *word-automatic* if, for some finite alphabet Σ, we have $S = \Sigma^*$, the relation $\rightarrow$ is a regular relation on S, and each $\lambda(P_i)$ is a regular subset of S. Likewise, a transition system $\mathcal{S}$ over AP is *tree-automatic* if, for some Υ and Σ, we have $S = \textsc{Tree}_\Upsilon(\Sigma)$, and all of $\rightarrow$ and $\lambda(P_i)$'s are regular tree relations/languages over $\textsc{Tree}_\Upsilon(\Sigma)$.

We measure the size of such a word- or tree-automatic transition system $\mathcal{S}$ as the total size of the transducer for $\rightarrow$, and the automata for S and $\lambda(P_i)$, for $P_i \in \text{AP}$. We shall assume that these are nondeterministic.

As mentioned already, pushdown systems and prefix-recognizable graphs are examples of word-automatic infinite transition systems, while PA-processes and graphs generated by ground tree rewrite systems are examples of tree-automatic transition systems.

3 Recurrent Reachability: The Word Case

We call a word-automatic transition system $\mathcal{S} = \langle S, \rightarrow, \lambda \rangle$ *transitive* if the relation $\rightarrow^+$ is regular. As we shall see shortly, if $\mathcal{S}$ is transitive, then the set $Rec(L)$ is regular too, for an arbitrary regular language L. This gives rise to two variants of the model-checking problem for recurrent reachability: in the global model-checking problem, we are given $\mathcal{S}$ and a language L represented by an NWA $\mathcal{A}$, and we want to construct an NWA accepting $Rec(L(\mathcal{A}))$. In the local version, we also have a word w, and we must check whether $w \in Rec(L(\mathcal{A}))$. That is,

GLOBAL MODEL-CHECKING:	INPUT: 1) A transitive word-automatic $\mathcal{S}$ 2) An NWA $\mathcal{A}$ OUTPUT: A description of $Rec(L(\mathcal{A}))$

LOCAL MODEL-CHECKING:	INPUT: 1) A transitive word-automatic $\mathcal{S}$ 2) An NWA $\mathcal{A}$ 3) a word w OUTPUT: *yes*, if $w \rightarrow^\omega L(\mathcal{A})$ *no*, otherwise

Throughout this section, we assume that the transition relation $\rightarrow$ of transitive $\mathcal{S}$ is given by a transducer $\mathcal{R}$, and that $\mathcal{R}^+$ is the transducer for $\rightarrow^+$ (which exists by the transitivity assumption). We shall also use the transducer for $\rightarrow^*$, denoted by $\mathcal{R}^*$. It can be obtained from $\mathcal{R}^+$ by letting it accept pairs $w \otimes w$.

Theorem 1. *Given a transitive word-automatic transition system $\mathcal{S} = \langle S, \rightarrow, \lambda \rangle$ and an NWA $\mathcal{A}$, the set $Rec(L(\mathcal{A}))$ of states w such that $w \rightarrow^\omega L(\mathcal{A})$ is regular.*

Moreover, if the transducer $\mathcal{R}^+$ for $\rightarrow^+$ is computable in time $t(|\mathcal{R}|)$, then one can compute an NWA recognizing $Rec(L(\mathcal{A}))$ of size $O(|\mathcal{R}^+|^2 \times |\mathcal{A}|)$ in time $t(|\mathcal{R}|) + O(|\mathcal{R}^+|^3 \times |\mathcal{A}|^2)$.

Corollary 2. *Given a transitive word-automatic transition system* $\mathcal{S} = \langle S, \rightarrow, \lambda \rangle$ *and an NWA* $\mathcal{A}$*, such that the transducer* $\mathcal{R}^+$ *is poly-time computable, both global and local model-checking for recurrent reachability are solvable in poly-time.*

As another corollary, consider the **EF**, **EX**-fragment of CTL, known as the EF-logic [23, 27]. Its formulae over AP $= \{P_1, \ldots, P_n\}$ are given by

$$\varphi, \varphi' := \top \mid P_i,\ i \leq n \mid \varphi \vee \varphi' \mid \neg\varphi \mid \mathbf{EX}\varphi \mid \mathbf{EF}\varphi.$$

Each formula, evaluated over a transition system $\mathcal{S} = \langle S, \rightarrow, \lambda \rangle$, defines a set $[\![\varphi]\!]_{\mathcal{S}} \subseteq S$ as follows:

(1) $[\![\top]\!]_{\mathcal{S}} = S$; (2) $[\![P_i]\!]_{\mathcal{S}} = \lambda(P_i)$;
(3) $[\![\varphi \vee \varphi']\!]_{\mathcal{S}} = [\![\varphi]\!]_{\mathcal{S}} \cup [\![\varphi']\!]_{\mathcal{S}}$; (4) $[\![\neg\varphi]\!]_{\mathcal{S}} = S - [\![\varphi]\!]_{\mathcal{S}}$;
(5) $[\![\mathbf{EX}\varphi]\!]_{\mathcal{S}} = \{s \mid \exists s' :\ s \rightarrow s' \text{ and } s' \in [\![\varphi]\!]_{\mathcal{S}}\}$;
(6) $[\![\mathbf{EF}\varphi]\!]_{\mathcal{S}} = \{s \mid \exists s' :\ s \rightarrow^* s' \text{ and } s' \in [\![\varphi]\!]_{\mathcal{S}}\}$.

If $\rightarrow^*$ is given by a regular transducer, then $[\![\varphi]\!]_{\mathcal{S}}$ is clearly effectively regular [6], and so the model-checking problem for EF-logic is decidable. We now extend this to the (EF+EGF)-logic, defined as the extension of EF-logic with the formulae $\mathbf{EGF}\varphi$ with the semantics

$$[\![\mathbf{EGF}\varphi]\!]_{\mathcal{S}} \ = \ Rec([\![\varphi]\!]_{\mathcal{S}}, \rightarrow^+) \ = \ \{s \mid s \rightarrow^\omega [\![\varphi]\!]_{\mathcal{S}}\}.$$

Theorem 1 extends decidability to (EF+EGF)-logic:

Corollary 3. *If* $\mathcal{S} = \langle S, \rightarrow, \lambda \rangle$ *is a transitive word-automatic transition system such that the transducer* $\mathcal{R}^+$ *is computable, then for each formula* φ *of (EF+EGF)-logic, the set* $[\![\varphi]\!]_{\mathcal{S}}$ *is regular, and an NWA defining* $[\![\varphi]\!]_{\mathcal{S}}$ *can be effectively constructed.*

We now prove Theorem 1. Throughout the proof, we let $\mathcal{M}$ stand for $\mathcal{R}^+$ and use unambiguous abbreviations such as $Rec(\mathcal{A}, \mathcal{M})$ for $Rec(L(\mathcal{A}), L(\mathcal{M}))$. By definition, we have $w \in Rec(\mathcal{A}, \mathcal{M})$ iff there exists a sequence $\{s_i\}_{i \in \mathbb{N}}$ of words with $s_0 = w$ such that $s_{i-1} \otimes s_i \in L(\mathcal{M})$ and $s_i \in L(\mathcal{A})$ for all $i > 0$. We now divide $Rec(\mathcal{A}, \mathcal{M})$ into two sets $Rec_1(\mathcal{A}, \mathcal{M})$ and $Rec_2(\mathcal{A}, \mathcal{M})$, where $Rec_1(\mathcal{A}, \mathcal{M})$ contains words with a witnessing infinite sequence $\{s_i\}_{i \in \mathbb{N}}$ that satisfies $s_j = s_k$ for some $j < k$, and $Rec_2(\mathcal{A}, \mathcal{M})$ contains words with a witnessing infinite sequence $\{s_i\}_{i \in \mathbb{N}}$ that satisfies $s_j \neq s_k$ for all distinct $j, k \in \mathbb{N}$. We shall write Rec_1 and Rec_2 when the intended automata $\mathcal{A}$ and $\mathcal{M}$ are clear from the context. Now notice that $Rec(L(\mathcal{A})) = Rec_1 \cup Rec_2$. It is easy to construct an NWA that recognizes Rec_1. Observe that, for all word w, we have $w \in Rec_1$ iff there exists a word w' such that $w \rightarrow^* w'$, $w' \rightarrow^+ w'$, and $w' \in L(\mathcal{A})$. By taking a product and then applying projection (e.g. see [6]), we can compute an NWA $\mathcal{A}_1$ that recognizes Rec_1 in time $O(|\mathcal{R}^+|^2 \times |\mathcal{A}||)$ with $|\mathcal{A}| = O(|\mathcal{R}^+|^2 \times |\mathcal{A}|)$.

Thus, it remains to construct the automaton $\mathcal{A}_2$ for Rec_2. We shall first compute a Büchi automaton $\mathcal{B}$ that recognizes an ω-word which represents the witnessing infinite sequence for membership in Rec_2. Once $\mathcal{B}$ is constructed, it is easy to obtain $\mathcal{A}_2$ as we shall see later. The most obvious representation of the

infinite sequence $\{s_i\}_{i\in\mathbb{N}}$ is $s_0 \otimes s_1 \otimes \ldots$. The problem with this representation is that it requires an infinite alphabet, and possibly infinitely many copies of the automata $\mathcal{A}$ and $\mathcal{M}$ to check whether $s_i \in L(\mathcal{A})$ and $s_{i-1} \rightarrow^+ s_i$ for all $i > 0$. Therefore, the first step towards solving the problem is to analyze the infinite witnessing paths and to show that it is sufficient to consider only infinite sequences of a special form. For the rest of this section, we let $\mathcal{A} = (Q_1, \delta_1, q_0^1, F_1)$ and $\mathcal{M} = \mathcal{R}^+ = (Q_2, \delta_2, q_0^2, F_2)$.

Lemma 4. *For every word $w \in \Sigma^*$, it is the case that $w \in Rec_2(\mathcal{A})$ iff there exist two infinite sequences $\{\alpha_i\}_{i\in\mathbb{N}}$ and $\{\beta_i\}_{i\in\mathbb{N}}$ of words over Σ such that*

1. *$\alpha_0 = w$ and $|\alpha_i| > 0$ for all $i > 0$,*
2. *$|\alpha_i| = |\beta_i|$ for all $i \in \mathbb{N}$,*
3. *there exists an infinite run r of $\mathcal{A}$ on $\beta_0\beta_1\ldots$ such that, for all $i \in \mathbb{N}$, the automaton $\mathcal{A}^q$ accepts α_{i+1}, where $q = r(|\beta_0 \ldots \beta_i|)$,*
4. *there exists an infinite run r' of $\mathcal{M}$ on $(\beta_0 \otimes \beta_0)(\beta_1 \otimes \beta_1)\ldots$ such that, for all $i \in \mathbb{N}$, $\mathcal{M}^q$ accepts $\alpha_i \otimes \beta_i\alpha_{i+1}$ where $q = r'(|\beta_0 \ldots \beta_{i-1}|)$.*

One direction of the lemma is easy: if 1)–4) hold, then from the infinite sequences $\{\alpha_i\}_{i\geq 0}$ and $\{\beta_i\}_{i\geq 0}$ we can form a new sequence $\{s_i\}_{i\geq 0}$ with $s_i := \beta_0 \ldots \beta_{i-1}\alpha_i$. Condition (3) ensures that $s_i \in L(\mathcal{A})$ for all $i > 0$, and condition (4) implies that $s_i \rightarrow^+ s_{i+1}$ for all $i \geq 0$. This implies that $w \in Rec_2(\mathcal{A})$ and thus proving sufficiency in Lemma 4.

The idea of the proof of Theorem 1 is that the sequences $\{\alpha_i\}_{i\geq 0}$ and $\{\beta_i\}_{i\geq 0}$ compactly represent a sequence $\{s_i\}_{i\geq 0}$ witnessing $w \in Rec_2(\mathcal{A})$. We shall later construct a Büchi automaton that recognizes precisely all ω-words of the form

$$(\alpha_0 \otimes \beta_0)\begin{bmatrix}\#\\\#\end{bmatrix}(\alpha_1 \otimes \beta_1)\begin{bmatrix}\#\\\#\end{bmatrix}(\alpha_2 \otimes \beta_2)\begin{bmatrix}\#\\\#\end{bmatrix}\ldots \qquad (*)$$

such that the sequences $\{\alpha_i\}_{i\geq 0}$ and $\{\beta_i\}_{i\geq 0}$ satisfy r.h.s. of Lemma 4. From such an automaton $\mathcal{B}$ it is easy to obtain an automaton recognizing $\alpha_0 = w \in Rec_2$.

Now we shall prove the other direction in Lemma 4: that the sequences $\{\alpha_i\}_{i\geq 0}$ and $\{\beta_i\}_{i\geq 0}$ exist under the assumption $w \in Rec_2(\mathcal{A})$. We will first need to extend the definition of $Rec_2(\mathcal{N}, \mathcal{T})$ to allow not necessarily transitive transducers $\mathcal{T}$: $w \in Rec_2(\mathcal{N}, \mathcal{T})$ iff there exists a sequence $\{s_i\}_{i\geq 0}$ of words such that $s_0 = w$, $s_i \neq s_{i'}$ for all distinct $i, i' \in \mathbb{N}$, $s_i \in L(\mathcal{N})$ for all $i > 0$, and $s_j \otimes s_k \in L(\mathcal{T})$ for all $k > j \geq 0$.

Lemma 5. *Suppose $\mathcal{N}$ and $\mathcal{T}$ are, respectively, an automaton and a transducer over Σ. For every word $w \in \Sigma^*$, if $w \in Rec_2(\mathcal{N}, \mathcal{T})$, then there exists a word $w'w''$ such that*

1. *$|w'| = |w|$ and $|w''| > 0$,*
2. *$w \otimes w'w'' \in L(\mathcal{T})$,*
3. *there exist an accepting run r of $\mathcal{N}$ on $w'w''$, and a run r' of $\mathcal{T}$ on $w' \otimes w'$ such that $w'' \in Rec_2(\mathcal{N}^{q_1}, \mathcal{T}^{q_1'})$, where $q_1 = r(|w|)$ and $q_1' = r'(|w|)$.*

Proof. Suppose that $w \in Rec_2(\mathcal{N}, \mathcal{T})$. Then, there exists an infinite sequence $\sigma = \{s_i\}_{i\in\mathbb{N}}$ such that $s_0 = w$, $s_i \neq s_{i'}$ for all distinct $i, i' \in \mathbb{N}$, and it is the case that, for all $i > 0$, the word s_i is in $L(\mathcal{N})$ with accepting run η_i, and for all distinct pair of indices $i' > i \geq 0$, we have $s_i \otimes s_{i'} \in L(\mathcal{T})$. As there are only finitely many different words of length $|w|$ but infinitely many different words in σ, we may assume that $|s_i| > |w|$ for all $i \geq 1$; for, otherwise, we may simply omit these words from σ. Now every word s_i, where $i > 0$, can be written as $s_i = u_i v_i$ for some words u_i, v_i such that $|u_i| = |w|$ and $|v_i| > 0$. As there are only finitely many different words of length $|w|$ and finitely many different runs of $\mathcal{N}$ of length $|w|$, by pigeonhole principle there must exist $k > 0$ such that $u_j = u_k$ and $\eta_j[0, |w|] = \eta_k[0, |w|]$ for infinitely many $j > 0$. Let $w' := u_k$ and $\eta := \eta_k[0, |w|]$. Therefore, we may discard all words s_i in σ with $i \geq 1$ such that $u_i \neq w'$ or η is not a prefix of η_i. By renaming indices, call the resulting sequence $\sigma = \{s_i\}_{i\in\mathbb{N}}$ and, for all $i \geq 1$, denote by η_i the accepting run of $\mathcal{N}$ on s_i that has η as a prefix. Notice that σ is still a witness for $w \in Rec_2(\mathcal{N}, \mathcal{T})$. So, let $\theta_{j,k}$, where $0 \leq j < k$, be the accepting run of $\mathcal{T}$ on $s_j \otimes s_k$. Let $\mathcal{C}$ be the *finite* set of all runs of $\mathcal{T}$ on $w' \otimes w'$. Notice that it is not necessarily the case that $|\mathcal{C}| = 1$ as $\mathcal{T}$ is nondeterministic. Consider the edge-labeled undirected graph $G = (V, \{E_u\}_{u\in\mathcal{C}})$ such that $V = \mathbb{Z}^+$ and

$$E_u = \{\{j, k\} : \ 0 < j < k \text{ and } u \text{ is a prefix of } \theta_{j,k} \ \}.$$

Notice that $\{E_u\}_{u\in\mathcal{C}}$ is a partition of $\{\{j, k\} : j \neq k, j, k > 0\}$, and so G is a complete graph. By (infinite) Ramsey theorem, G has a monochromatic complete infinite subgraph $H = (V', E_u)$ for some $u \in \mathcal{C}$. Set $r' := u$. Notice that if V' contains the elements $i_1 < i_2 < \ldots$, then θ_{i_j,i_k} with $k > j \geq 1$ has u as a prefix. Therefore, we can discard all words s_i $(i > 0)$ in σ such that $i \notin V'$ and by renaming indices call the resulting sequence $\sigma = \{s_i\}_{i\in\mathbb{N}}$. We also adjust the sequence $\{\eta_i\}_{i>0}$ of accepting runs by omitting the appropriate runs and adjusting indices. We now set w'' to be the unique word v such that $s_1 = w'v$. It is easy to see that (1) and (2) are satisfied. Setting $r = \eta_1$, it is easy to check that $w'' \in Rec_2(\mathcal{N}^{q_1}, \mathcal{T}^{q_1'})$ with witnessing sequence $\{t_i\}_{i>0}$, where t_i is the unique word such that $s_i = w't_i$ for all $i > 0$. □

Now it is not difficult to inductively construct the desired sequences $\{\alpha_i\}_{i\geq 0}$ and $\{\beta_i\}_{i\geq 0}$ by using lemma 5 at every induction step. The gist of the proof is that from the word $w'w''$ given by lemma 5 at induction step k, we will set $\beta_k = w'$, $\alpha_{k+1} = w''$, and extend the partial runs r and r' in lemma 4. Notice that we now have $w'' \in Rec_2(\mathcal{N}^{q_1}, \mathcal{T}^{q_1'})$, which sets up the next induction step. See full version for a detailed argument. This completes the proof of lemma 4.

Now we construct a Büchi automaton $\mathcal{B}$ accepting ω-words of the form $(*)$, where α_i's and β_i's are given by Lemma 4. We first give an informal description of how to implement $\mathcal{B}$. The automaton $\mathcal{B}$ will attempt to guess the runs r and r', while at the same time checking that the runs satisfy conditions 3–4 in Lemma 4. To achieve this, $\mathcal{B}$ will run a copy of $\mathcal{A}$ and $\mathcal{M}$, while simultaneously also running a few other copies of $\mathcal{A}$ and $\mathcal{M}$ to check that the runs r and r' guessed so far satisfy conditions 3) and 4) along the way. The automaton $\mathcal{B}$

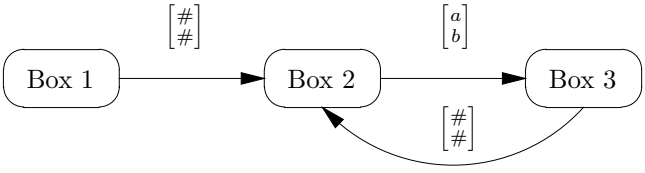


Fig. 1. A bird's eye view of the Büchi automaton $\mathcal{B}$

consists of three components depicted as Boxes 1, 2, and 3 in Figure 1. The first box is used for reading the prefix of the input before the first occurrence of $\left[\begin{smallmatrix}\#\\\#\end{smallmatrix}\right]$, while the other boxes are used for reading the remaining suffix. Boxes 2–3 are essentially identical, i.e., they have the same sets of states and essentially the same transition functions. When $\mathcal{B}$ arrives in Box 2, it will read a single letter in Σ^2 and goes to Box 3 so as to make sure that $|\alpha_i| > 0$ for each $i > 0$. When $\mathcal{B}$ is in Box 3, it will go to Box 2 upon reading the letter $\left[\begin{smallmatrix}\#\\\#\end{smallmatrix}\right]$. We will set all states in Box 2 as the final states so as to make sure that infinitely many $\left[\begin{smallmatrix}\#\\\#\end{smallmatrix}\right]$ is seen, i.e., the sequences $\{\alpha_i\}_i$ and $\{\beta_i\}_i$ are both infinite, and each words α_i and β_i are finite.

More formally, the automaton $\mathcal{B} = (\Sigma^2 \cup \{\left[\begin{smallmatrix}\#\\\#\end{smallmatrix}\right]\}, Q, \delta, q_0, F)$ is defined as follows. We set $Q := (Q_1 \times Q_2 \times Q_2) \uplus (Q_1 \times Q_2 \times Q_1 \times Q_2 \times Q_2 \times \{1, 2\})$, where $Q_1 \times Q_2 \times Q_2$ are the states in box #1, and $Q_1 \times Q_2 \times Q_1 \times Q_2 \times Q_2 \times \{i\}$ are states in box $\#(i+1)$. The initial state is $q_0 := (q_0^1, q_0^2, q_0^2)$. The first and the last components in each state are meant for guessing the infinite runs r and r'. The second component of each state in box #1 is used for guessing a prefix of the accepting run of $\mathcal{M}$ on $\alpha_0 \otimes \beta_0\alpha_1$. The automaton $\mathcal{B}$ will finish this guessing when it reaches box #3 upon the completion of parsing $\alpha_1 \otimes \beta_1$. When $\mathcal{B}$ is presently in box #2 or #3 and reading $\alpha_i \otimes \beta_i$, where $i > 0$, the third and fourth components of the states are used for checking that $\beta_0 \dots \beta_{i-1}\alpha_i \in L(\mathcal{A})$ and $\beta_0 \dots \beta_{i-2}\alpha_{i-1} \otimes \beta_0 \dots \beta_{i-1}\alpha_i \in L(\mathcal{M})$, respectively. At the same time, the second component will be checking that $\beta_0 \dots \beta_{i-1}\alpha_i \otimes \beta_0 \dots \beta_i\alpha_{i+1} \in L(\mathcal{M})$, which will be completed in the next iteration. We now formally define the transition function. We set

$$\delta((q, q', q''), \left[\begin{smallmatrix}a\\b\end{smallmatrix}\right]) := \begin{cases} \delta_1(q, b) \times \delta_2(q', \left[\begin{smallmatrix}a\\b\end{smallmatrix}\right]) \times \delta_2(q'', \left[\begin{smallmatrix}b\\b\end{smallmatrix}\right]) & \text{, if } a, b \neq \# \\ (q, q'', q, q', q'', 1) & \text{, if } a = b = \# \\ \emptyset & \text{, otherwise.} \end{cases}$$

and, when $\mathcal{B}$ is in a state in $Q_1 \times Q_2 \times Q_1 \times Q_2 \times Q_2 \times \{i\}$, where $i = 1, 2$, we define

$$\delta((q_1, q_2, q_1', q_2', q_2'', i), \left[\begin{smallmatrix}a\\b\end{smallmatrix}\right]) := \delta_1(q_1, b) \times \delta_2(q_2, \left[\begin{smallmatrix}a\\b\end{smallmatrix}\right]) \times \delta_1(q_1', a) \times \delta_2(q_2', \left[\begin{smallmatrix}\perp\\a\end{smallmatrix}\right]) \times \delta_2(q_2'', \left[\begin{smallmatrix}b\\b\end{smallmatrix}\right]) \times \{2\}$$

if $a, b \neq \#$. If $q_1' \in F_1$, and $q_2' \in F_2$, then we set

$$\delta((q_1, q_2, q_1', q_2', q_2'', 2), \begin{bmatrix}\#\\\#\end{bmatrix}) = (q_1, q_2'', q_1, q_2, q_2'', 1).$$

Finally, the set of final states are $F := Q_1 \times Q_2 \times Q_1 \times Q_2 \times Q_2 \times \{1\}$. It is easy to see that $\mathcal{B}$, as claimed, recognizes exactly ω-words of the word of the form $(*)$ such that the sequences $\{\alpha_i\}_{i\in\mathbb{N}}$ and $\{\beta_i\}_{i\in\mathbb{N}}$ satisfy the conditions in Lemma 4.

Now, from $\mathcal{B}$ we can easily compute the automaton $\mathcal{A}_2 = (Q', \Sigma, \delta', q_0', F')$ that recognizes Rec_2. Roughly speaking, the automaton $\mathcal{A}_2$ will accept the set of finite words α_0 such that there exist two sequences $\{\alpha_i\}_{i>0}$ and $\{\beta_i\}_{i\geq 0}$ such that the ω-word $(*)$ is accepted by $\mathcal{B}$. Therefore, we will set the new set of states Q' to be $Q_1 \times Q_2 \times Q_2$, i.e., the first component of $\mathcal{B}$ in Fig. 1. We apply projection operation on the transition function δ of $\mathcal{B}$ to obtain δ'. More formally, if $a \in \Sigma$, we set

$$\delta'((q_1, q_2, q_2'), a) = \bigvee_{b\in\Sigma} \delta((q_1, q_2, q_2'), \begin{bmatrix}a\\b\end{bmatrix}).$$

Finally, the new set F' of final states will those states in Q' from which $\mathcal{B}$ can accept some ω-words of the form $\begin{bmatrix}\#\\\#\end{bmatrix} w$ for some ω-word w. For this, we can apply the standard algorithm for testing nonemptiness for Büchi automata, which takes linear time. Theorem 1 is now immediate. □

Application: Pushdown systems. We shall use the definition of [10, 11, 22], which subsumes a more common definition of [14, 15, 27] based on configurations of pushdown automata and transitions between them. A *pushdown system* over the alphabet Σ is given by a finite set Δ of rules of the form $u \to v$ where $u, v \in \Sigma^*$. Let $\mathrm{Dom}(\Delta)$ denote the set of words u for which there is a rule $u \to v$ in Δ. Then Δ generates a relation $\to_\Delta$ over Σ^* as follows: $(w, w') \in \mathcal{R}_\Delta$ iff there exist $x, u, v \in \Sigma^*$ such that $w = xu$, $w' = xv$, and $u \to v$ is a rule in Δ. We thus compute recurrent reachability over pushdown systems $\langle \Sigma^*, \to_\Delta, \lambda \rangle$.

The binary relation $\to_\Delta$ is regular, and can be given by a transducer $\mathcal{R}_\Delta$ whose size is linear in $\|\Delta\|$ (where $\|\Delta\|$ is the sum of the lengths of each word in Δ). Caucal [10] proved that, for each pushdown system Δ, the relation $\to_\Delta^*$ is a poly-time-computable rational transduction[1]. Later in [11] he noted that the given transducer is also regular. For completeness, we sketch how his construction gives a regular transducer $\mathcal{R}_\Delta^*$ for $\to_\Delta^*$ in poly-time. Recall the following well-known proposition, which is proven using the standard "saturation" construction (e.g. see [8, 10, 14]).

Proposition 6. *Given a pushdown system Δ and a nondeterministic automaton $\mathcal{A}$, one can compute two automata $\mathcal{A}_{pre^*}$ and $\mathcal{A}_{post^*}$ for $pre^*(L(\mathcal{A}))$ and $post^*(L(\mathcal{A}))$ in poly-time.*

In fact, the algorithm given in [14] computes the automata in cubic time, and the sizes of $\mathcal{A}_{pre^*}$ and $\mathcal{A}_{post^*}$ are at most quadratic in $|\mathcal{A}|$. To construct $\mathcal{R}_\Delta^*$ using this proposition, we shall need the following easy lemma.

[1] Rational transducers are strictly more powerful than regular transducers.

Lemma 7 ([10]). *Given a pushdown system Δ and two words $u, v \in \Sigma^*$, then $u \rightarrow_\Delta^* v$ iff there exist words $x, y, z \in \Sigma^*$ and word $w \in Dom(\Delta) \cup \{\varepsilon\}$ such that $u = xy$, $v = xz$, $y \rightarrow_\Delta^* w$, and $w \rightarrow_\Delta^* z$.*

Now constructing $\mathcal{R}_\Delta^*$ is easy. First, we use Proposition 6 to compute the automata $\mathcal{A}^w_{pre^*}$ and $\mathcal{A}^w_{post^*}$ that recognize $pre^*(w)$ and $post^*(w)$ for every $w \in \mathrm{Dom}(\Delta) \cup \{\varepsilon\}$. Then, on input $u \otimes v$, the transducer guesses a word $w \in \mathrm{Dom}(\Delta) \cup \{\varepsilon\}$ and a position at which the prefix x in Lemma 7 ends, and then simultaneously runs the automata $\mathcal{A}^w_{pre^*}$ and $\mathcal{A}^w_{post^*}$ to verify that the top part y and the bottom part z of the remaining input word (preceding the $\perp$ symbol) satisfy $y \in L(\mathcal{A}^w_{pre^*})$ and $z \in L(\mathcal{A}^w_{post^*})$. We thus obtain a transducer $\mathcal{R}^*$ of size $O(\|\Delta\|^2)$. By taking a product, we compute a transducer $\mathcal{R}^+$ of size $O(\|\Delta\|^3)$ in poly-time. Therefore, Theorem 1 implies the following.

Theorem 8. *Both global and local model-checking for recurrent reachability over pushdown systems are solvable in poly-time.*

That is, for a pushdown system Δ and a nondeterministic automaton $\mathcal{A}$ over an alphabet Σ, one can compute in polynomial time an NWA recognizing $Rec(L(\mathcal{A}), \rightarrow_\Delta^+)$.

4 Recurrent Reachability: The Tree Case

Recall that in a tree-automatic transition system $\mathcal{S} = (S, \rightarrow, \lambda)$, the relation $\rightarrow$ and the sets $\lambda(P_i)$'s are given as tree automata. As in the word case, such a transition system is said to be *transitive* if the relation $\rightarrow^+$ is regular. We now extend our results from Section 3 to transitive tree-automatic transition systems.

Theorem 9. *Given an NTA $\mathcal{A}$ and a transitive tree-automatic transition system $\mathcal{S} = \langle S, \rightarrow, \lambda \rangle$, the set $Rec(L(\mathcal{A}))$ of states $T \in S$ such that $T \rightarrow^\omega L(\mathcal{A})$ is regular. Moreover, if the transducer $\mathcal{R}^+$ for $\rightarrow^+$ is computable in time $t(|\mathcal{R}|)$, then one can compute an NTA recognizing $Rec(L(\mathcal{A}))$ of size $O(|\mathcal{R}^+|^2 \times |\mathcal{A}|)$ in time $t(|\mathcal{R}|) + O(|\mathcal{R}^+|^6 \times |\mathcal{A}|^4)$.*

As in the word case, this implies two corollaries:

Corollary 10. *If $\mathcal{S}$ is transitive and tree-automatic and $\mathcal{R}^+$ is poly-time computable, then both global and local model-checking for recurrent reachability are solvable in poly-time.*

Corollary 11. *If $\mathcal{S} = \langle S, \rightarrow, \lambda \rangle$ is a transitive tree-automatic transition system such that the transducer $\mathcal{R}^+$ is computable, then for each formula φ of (EF+EGF)-logic, the set $[\![\varphi]\!]_\mathcal{S}$ is regular, and an NTA defining $[\![\varphi]\!]_\mathcal{S}$ can be effectively constructed.*

The proof follows the same basic steps as the proof of Theorem 1: we first show that it is sufficient to consider only infinite witnessing sequences that have a representation as an infinite tree over a finite labeling alphabet; we then construct

a tree automaton (over infinite trees) with a Büchi acceptance condition that recognizes such sequences; and from such an automaton we construct a NTA for $Rec(L(\mathcal{A}))$ by applying projection and checking nonemptiness for Büchi tree automata. As checking nonemptiness for Büchi tree automata is quadratic [26] instead of linear as in the word case, the degree of the polynomials in Theorem 9 doubles. While all steps are similar to those in the word case, there are many technical differences; in particular in the coding of an infinite sequence by a single infinite tree. See full version for details of the proof.

Application: Ground tree rewrite systems. Ground tree rewrite systems have been intensely studied in the rewriting, automata, and verification communities [12, 13, 18, 19]. We now show that a result by Löding [19] on poly-time model-checking for recurrent reachability and decidability of model checking (EF+EGF)-logic over such systems, which was proved with a specialized method for RGTRSs, can be obtained as a corollary of Theorem 9.

A *ground tree rewrite system* (GTRS) over Σ-labeled Υ-trees is a finite set Δ of transformation rules of the form $t \to t'$ where $t, t' \in \textsc{Tree}_\Upsilon(\Sigma)$. If we permit rules of the form $L \to L'$, where L and L' are tree languages given by some NTAs, then we call Δ a *regular ground tree rewrite system* (RGTRS). Obviously, RGTRSs generalize GTRSs. We define $\|\Delta\|$ as the sum of the sizes of automata in Δ. The RGTRS Δ also generates a binary relation $\to_\Delta$ over $\textsc{Tree}_\Upsilon(\Sigma)$: For a tree T and a node u in it, let T_u be the subtree of T rooted at u. Given two trees T and T', we let $T \to_\Delta T'$ iff there exists a node u in T and a rule $L \to L'$ in Δ such that $T_u \in L$ and $T' = T[t'/u]$ for some $t' \in L'$, where $T[t'/u]$ is the tree obtained from T by replacing the node u by the tree t'.

Given Δ, it is easy to compute a tree transducer $\mathcal{R}_\Delta$ for $\to_\Delta$ in time $O(\|\Delta\|)$; it guesses a node u in the input tree $T \otimes T'$ and a rule in Δ to apply at u in T to obtain T'. The following has been proven in [13] and [12, chapter 3].

Proposition 12. *Given a RGTRS Δ, the transitive closure relation $\to_\Delta^+$ is regular, and a transducer defining it can be computed in time polynomial in $|\mathcal{R}_\Delta|$.*

In fact, the proof for the above proposition constructs "ground tree transducers", which are a subclass of the notion of transducers we are considering in this paper (e.g. see [12, chapter 3]).

Combining this proposition with corollaries 10 and 11, we obtain:

Corollary 13. *(Löding [19]) Both global and local model checking for recurrent reachability over RGTRSs are solvable in poly-time. Model checking (EF+EGF)-logic over RGTRSs with regular atomic predicates is decidable.*

Application: PA-processes. PA [4, 22] is a well-known process algebra allowing sequential and parallel compositions, but no communication. It generalizes basic parallel processes (BPP), and context-free processes (BPA), but is incomparable to pushdown processes and Petri nets (e.g. see [22]). PA has found applications in the interprocedural dataflow analysis of parallel programs [16].

We review the basic definitions, following the presentation of [20]: we initially distinguish terms that are equivalent up to simplification laws. The definition of PA usually includes transition labels, which we omit to simplify our presentation (however, the results easily hold when we incorporate transition labels). Fix a finite set $Var = \{X, Y, Z, \ldots\}$ of process variables. *Process terms* over Var, denoted by $\mathcal{F}_{Var}$, are generated by the grammar:

$$t, t' \;:=\; 0 \;\mid\; X, \quad X \in Var \;\mid\; t.t' \;\mid\; t \| t'$$

where 0 denotes a "nil" process, and $t.t'$ and $t\|t'$ are sequential and parallel compositions, resp. Process terms can be viewed as Σ-labeled binary trees, where $\Sigma = Var \cup \{0, \|, \cdot\}$. In particular, inner nodes are always labeled by '.' or '$\|$', while leaves are labeled by elements in $Var \cup \{0\}$. A PA *declaration* over $\mathcal{F}_{Var}$ is a finite set Δ of rewrite rules of the form $X \to t$, where $X \in Var$, and $t \in \mathcal{F}_{Var}$. We set $\mathrm{Dom}(\Delta) = \{X : (X \to t) \in \Delta, \text{ for some } t \in \mathcal{F}_{Var}\}$, and $Var_\emptyset = Var - \mathrm{Dom}(\Delta)$. The set Δ generates a transition relation $\to_\Delta$ on process terms defined by:

$$\frac{t_1 \to t_1'}{t_1\|t_2 \to t_1'\|t_2} \qquad \frac{t_1 \to t_1'}{t_1.t_2 \to t_1'.t_2} \qquad \frac{}{X \to t}\ (X \to t) \in \Delta$$

$$\frac{t_2 \to t_2'}{t_1\|t_2 \to t_1\|t_2'} \qquad \frac{t_2 \to t_2'}{t_1.t_2 \to t_1.t_2'}\ t_1 \in \mathrm{IsNil}$$

Here IsNil is the set of "terminated" process terms, i.e., those in which all variables are in $Var_\emptyset$. It is easy to see that there is a regular transducer $\mathcal{R}_\Delta$ over process terms for $\to_\Delta$, whose size is linear in the size $\|\Delta\|$ of Δ. It is defined in in the same way as for GTRSs, except that when it guesses a leaf node at which a rule is applied, it must further ensure that v has no '.'-labeled ancestor u such that v is a descendant $u1$ and that T_{u0} is not a terminated process term.

Theorem 14 ([16, 20, 21]). *Given a PA declaration Δ and a NTA $\mathcal{A}$ describing a set of process terms over Var, the sets $pre^*(L(\mathcal{A}))$ and $post^*(L(\mathcal{A}))$ are regular, for which NTAs can be computed in time $O(\|\Delta\| \times |\mathcal{A}|)$, and one can construct a regular transducer $\mathcal{R}^+$ for $\to^+$ in poly-time*[2].

We consider only languages and atomic propositions that are interpreted as regular subsets of $\mathcal{F}_{Var}$. This poses no problem as $\mathcal{F}_{Var}$ is easily seen a regular subset of $\mathrm{TREE}_2(\Sigma)$ and no tree $t \in \mathcal{F}_{Var}$ is related by $\to$ to a tree $t' \in \mathrm{TREE}_2(\Sigma) - \mathcal{F}_{Var}$. From Theorem 14 and Corollaries 10 and 11, we obtain:

Theorem 15. *Both global and local model checking for recurrent reachability over* **PA** *are solvable in poly-time. Model checking (EF+EGF)-logic over PA is decidable.*

In the study of PA processes, it is common to use a structural equivalence on process terms. We now extend our results to PA modulo structural equivalence.

[2] Lugiez and Schnoebelen first proved this in [20] for a more general notion of transducers, but later in [21] realized that regular transducers suffice.

Let $\equiv$ be the smallest equivalence relation on $\mathcal{F}_{Var}$ that satisfies the following:

$$t.0 \equiv t \qquad 0.t \equiv t \qquad t\|0 \equiv t \qquad t\|t' \equiv t'\|t$$
$$(t\|t')\|t'' \equiv t\|(t'\|t'') \qquad (t.t').t'' \equiv t.(t'.t'')$$

We let $[t]_\equiv$ stand for the equivalence class of t and $[L]_\equiv$ for $\bigcup_{t\in L}[t]_\equiv$. We write $L/\equiv$ for $\{[t]_\equiv |\ t \in L\}$. It was shown in [20] that, for each $t \in \mathcal{F}_{Var}$, $[t]_\equiv$ is a regular tree language, although the set $[L]_\equiv$ need not be regular even for regular L. Given a PA declaration Δ, the equivalence $\equiv$ generates a transition relation $[t]_\equiv \Rightarrow [u]_\equiv$ over $\mathcal{F}_{Var}/\equiv$ which holds iff there exist $t' \in [t]_\equiv$ and $u' \in [u]_\equiv$ such that $t' \to u'$. We need the following result:

Lemma 16 ([20]). *The relation $\equiv$ is bisimulation: for all $t, t', u \in \mathcal{F}_{Var}$, if $t \equiv t'$ and $t \to u$, then there exists $u' \in \mathcal{F}_{Var}$ such that $t' \to u'$ and $u \equiv u'$.*

Now it is not hard to show that, for every NTA $\mathcal{A}$, the set $Rec(L(\mathcal{A}))$ is closed under $\equiv$, if $L(\mathcal{A})$ is closed under $\equiv$. This also implies that $Rec(L(\mathcal{A})) = [Rec(L(\mathcal{A}))]_\equiv = \{t : t \in Rec(L(\mathcal{A})/\equiv, \Rightarrow^+)\}$. In the following, we consider only languages that are closed under $\equiv$.

Theorem 17. *Given an NTA $\mathcal{A}$ such that $L(\mathcal{A})$ is closed under $\equiv$ and a process term $t \in \mathcal{F}_{Var}$, it is possible to decide whether $[t]_\equiv \Rightarrow^\omega L(\mathcal{A})/\equiv$ in PTIME.*

Since $Rec(L(\mathcal{A})) = [Rec(L(\mathcal{A}))]_\equiv$, we need only compute an NTA for $Rec(L(\mathcal{A}))$ and test whether $t \in Rec(L(\mathcal{A}))$. These can be done in PTIME by theorem 15.

We now move to model checking (EF+EGF)-logic over PA modulo $\equiv$. Suppose $\mathcal{S} = \langle S, \to, \lambda\rangle$ is a transition system generated by some PA-declaration and that each $\lambda(P)$ is closed under $\equiv$. In fact, the standard atomic propositions for PA-processes include sets of process terms of the form $[t]_\equiv$ and *action-based predicates*, i.e., sets of all terms t in which some transitions in Δ can be applied (and these are obviously closed under $\equiv$ and of size $O(\|\Delta\|)$). Now Lemma 16 implies that $[\![\varphi]\!]_\mathcal{S}$ is closed under $\equiv$ for (EF+EGF)-formulae φ, and we obtain:

Theorem 18. *The problem of model checking for (EF+EGF)-logic over PA modulo $\equiv$ is decidable whenever all atomic propositions are closed under $\equiv$.*

5 Future Work

We mention some possible future work. We would like to further study algorithmic improvements of our general technique, e.g., in its current form it gives a polynomial of degree higher than the specialized technique of [19] for RGTRSs. We would also like to investigate stronger but nonrestrictive conditions that ensure decidability of stronger logics (e.g. CTL*) within our framework; it is easy to show that our current condition is insufficient. Finally, we would like to study when our technique could generate elementary complexity algorithms for (EF+EGF)-logic, or just EF-logic alone. This problem is still open even for PA-processes and GTRSs [19, 23].

Acknowledgements. We thank Richard Mayr and anonymous referees for their helpful comments. The authors were supported by EPSRC grant E005039, the second author also by EC grant MEXC-CT-2005-024502.

References

1. Abdulla, P.A., Jonsson, B.: Undecidable verification problems for programs with unreliable channels. Inf. Comput. 130(1), 71–90 (1996)
2. Abdulla, P.A., Jonsson, B., Mahata, P., d'Orso, J.: Regular tree model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 555–568. Springer, Heidelberg (2002)
3. Abdulla, P.A., Jonsson, B., Nilsson, M., Saksena, M.: A survey of regular model checking. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 35–48. Springer, Heidelberg (2004)
4. Baeten, J., Weijland, W.: Process Algebra. In: CUP (1990)
5. Benedikt, M., Libkin, L., Neven, F.: Logical definability and query languages over ranked and unranked trees. ACM Trans. Comput. Logic 8(2), 11 (2007)
6. Blumensath, A., Grädel, E.: Automatic structures. In: LICS 2000, pp. 51–60 (2000)
7. Boigelot, B., Legay, A., Wolper, P.: Iterating transducers in the large. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 223–235. Springer, Heidelberg (2003)
8. Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: Application to model-checking. In: Mazurkiewicz, A., Winkowski, J. (eds.) CONCUR 1997. LNCS, vol. 1243, pp. 135–150. Springer, Heidelberg (1997)
9. Bouajjani, A., Legay, A., Wolper, P.: Handling liveness properties in (ω-)regular model checking. In: INFINITY, pp. 101–115 (2004)
10. Caucal, D.: On the regular structure of prefix rewriting. In: Arnold, A. (ed.) CAAP 1990. LNCS, vol. 431, pp. 61–86. Springer, Heidelberg (1990)
11. Caucal, D.: On the regular structure of prefix rewriting. Theor. Comput. Sci. 106(1), 61–86 (1992)
12. Comon, H., et al.: Tree Automata: Techniques and Applications (2007), http://www.grappa.univ-lille3.fr/tata
13. Dauchet, M., Tison, S.: The theory of ground rewrite systems is decidable. In: LICS 1990, pp. 242–248 (1990)
14. Esparza, J., Hansel, D., Rossmanith, P., Schwoon, S.: Efficient algorithms for model checking pushdown systems. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 232–247. Springer, Heidelberg (2000)
15. Esparza, J., Kucera, A., Schwoon, S.: Model checking LTL with regular valuations for pushdown systems. Inf. Comput. 186(2), 355–376 (2003)
16. Esparza, J., Podelski, A.: Efficient algorithms for pre* and post* on interprocedural parallel flow graphs. In: POPL 2000, USA, pp. 1–11. ACM, New York (2000)
17. Jonsson, B., Nilsson, M.: Transitive closures of regular relations for verifying infinite-state systems. In: Schwartzbach, M.I., Graf, S. (eds.) TACAS 2000. LNCS, vol. 1785, pp. 220–234. Springer, Heidelberg (2000)
18. Löding, C.: Model-checking infinite systems generated by ground tree rewriting. In: Nielsen, M., Engberg, U. (eds.) FOSSACS 2002. LNCS, vol. 2303, pp. 280–294. Springer, Heidelberg (2002)
19. Löding, C.: Reachability problems on regular ground tree rewriting graphs. Theory Comput. Syst. 39(2), 347–383 (2006)

20. Lugiez, D., Schnoebelen, P.: The regular viewpoint on PA-processes. Theor. Comput. Sci. 274(1-2), 89–115 (2002)
21. Lugiez, D., Schnoebelen, P.: Decidable first-order transition logics for PA-processes. Inf. Comput. 203(1), 75–113 (2005)
22. Mayr, R.: Process rewrite systems. Inf. Comput. 156(1-2), 264–286 (2000)
23. Mayr, R.: Decidability of model checking with the temporal logic EF. Theor. Comp. Sci. 256(1-2), 31–62 (2001)
24. Thomas, W.: Constructing infinite graphs with a decidable MSO-theory. In: Rovan, B., Vojtáš, P. (eds.) MFCS 2003. LNCS, vol. 2747, pp. 113–124. Springer, Heidelberg (2003)
25. Vardi, M.Y.: An automata-theoretic approach to linear temporal logic. In: Proc. Banff Higher-Order Workshop, pp. 238–266
26. Vardi, M.Y., Wolper, P.: Automata-theoretic techniques for modal logics of programs. JCSS 32(2), 183–221 (1986)
27. Walukiewicz, I.: Model checking CTL properties of pushdown systems. In: Kapoor, S., Prasad, S. (eds.) FST TCS 2000. LNCS, vol. 1974, pp. 127–138. Springer, Heidelberg (2000)

Alternation Elimination by Complementation* (Extended Abstract)**

Christian Dax and Felix Klaedtke

ETH Zurich, Computer Science Department, Switzerland

Abstract. In this paper, we revisit constructions from the literature that translate alternating automata into language-equivalent nondeterministic automata. Such constructions are of practical interest in finite-state model checking, since formulas of widely used linear-time temporal logics with future and past operators can directly be translated into alternating automata. We present a construction scheme that can be instantiated for different automata classes to translate alternating automata into language-equivalent nondeterministic automata. The scheme emphasizes the core ingredient of previously proposed alternation-elimination constructions, namely, a reduction to the problem of complementing nondeterministic automata. Furthermore, we clarify and improve previously proposed constructions for different classes of alternating automata by recasting them as instances of our construction scheme. Finally, we present new complementation constructions for 2-way nondeterministic automata from which we then obtain novel alternation-elimination constructions.

1 Introduction

Alternating automata are a powerful tool in finite-state model checking. Here, they serve as a glue between declarative specification languages like LTL [26] and PSL [1] and simple graph-like structures such as nondeterministic Büchi automata, which are well suited for algorithmic treatment, see e.g., [31]. By establishing translations from alternating automata to nondeterministic Büchi automata, one reduces the model checking problem for finite-state systems to a reachability problem on simple graph-like structures, see e.g., [13]. Similarly, such translations can be used to solve the satisfiability problem for declarative specification languages like LTL and PSL.

Translations of declarative specification languages into alternating automata are usually rather direct and easy to establish due to the rich combinatorial structure of alternating automata. Translating an alternating automaton into a nondeterministic Büchi automaton is a purely combinatorial problem. Hence, using alternating automata as an intermediate step is a mathematically elegant way to formalize such translations and to establish their correctness. Another

* Supported by the Swiss National Science Foundation (SNF).

** Due to space limitations, some proofs have been omitted. These can be found in an extended version of the paper, which is available from the authors' webpages.

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 214–229, 2008.

more practical advantage of such translations is that several automata-based techniques are applicable to optimize the outcome of such translations, e.g., simulation-based reduction techniques [9, 10].

Different classes of alternating automata are used for these kinds of translations depending on the expressive power of the specification language. For instance, for LTL, a restricted class of alternating automata suffices, namely the so-called very-weak alternating Büchi automata [21, 27]. These restrictions have been exploited to obtain efficient translators from LTL to nondeterministic Büchi automata, see [11]. For more expressive languages like the linear-time μ-calculus μLTL [2, 28], one uses alternating parity automata, and for fragments of the standardized property specification language PSL [1], one uses alternating Büchi automata [5]. If the temporal specification language has future and past operators, one uses 2-way alternating automata instead of 1-way alternating automata, see, e.g., [12, 15, 30]. Due to the immediate practical relevance in finite-state model checking, different constructions have been developed and implemented for translating a given alternating automaton into a language-equivalent nondeterministic automaton like the ones mentioned above.

In this paper, we present a general construction scheme for translating alternating automata into language-equivalent nondeterministic automata. In a nutshell, the general construction scheme shows that the problem of translating an alternating automaton into a language-equivalent nondeterministic automaton reduces to the problem of complementing a nondeterministic automaton. We also show that the nondeterministic automaton that needs to be complemented inherits structural and semantic properties of the given alternating automaton. We exploit these inherited properties to optimize the complementation constructions for special classes of alternating automata.

Furthermore, we instantiate the construction scheme to different classes of alternating automata. Some of the constructions that we obtain share similar technical details with previously proposed constructions as, e.g., the ones described in [11, 17, 22]. Some of them even produce the same nondeterministic Büchi automata modulo minor technical details. However, recasting these known constructions in such a way that they become instances of the construction scheme increases their accessibility. In particular, correctness proofs become modular and less involved. Another benefit of utilizing the construction scheme is that differences and similarities between the translations for the different classes of alternating automata become apparent.

We also present novel alternation-elimination constructions. These constructions are instances of our construction scheme and utilize a new complementation construction for so-called loop-free 2-way nondeterministic co-Büchi automata. In particular, we obtain an alternation-elimination construction that translates a loop-free 2-way alternating Büchi automaton with n states into a language-equivalent nondeterministic Büchi automaton with at most $O(2^{4n})$ states. This construction has potential applications for translating formulas from fragments of PSL extended with temporal past operators into nondeterministic Büchi automata. To the best of our knowledge, the best known construction for this class

of alternating automata results in nondeterministic Büchi automata of size at most $2^{O(n^2)}$ [15].

Overall, we see our contributions as twofold. On the one hand, the presented general construction scheme extracts and uniformly identifies essential ingredients for translating various classes of alternating automata into language-equivalent nondeterministic ones. Previously proposed alternation-elimination constructions for several classes of alternating automata, e.g. [11, 12, 15, 25, 28, 30] are based on similar ingredients. On the other hand, we clarify and improve existing alternation-elimination constructions for different classes of alternating automata, and we provide novel ones.

We proceed as follows. In Section 2, we give background on alternating automata. In Section 3, we give the general construction scheme. In Section 4, we present instances of that construction scheme for different classes of alternating automata and revisit previously proposed alternation-elimination constructions. Finally, in Section 5, we draw conclusions.

2 Background

We assume that the reader is familiar with automata theory. In this section, we recall the relevant background in this area and fix the notation used throughout this paper.

Given an alphabet Σ, Σ^* is the set of finite words over Σ and Σ^ω is the set of infinite words over Σ. Let w be a word over Σ. We denote its length by $|w|$. Note that $|w| = \infty$ if $w \in \Sigma^\omega$. For $i < |w|$, w_i denotes the ith letter of w, and we write w^i for the word $w_0 w_1 \dots w_{i-1}$, where $i \in \mathbb{N} \cup \{\infty\}$ with $i \leq |w|$. The word $u \in \Sigma^* \cup \Sigma^\omega$ is a *prefix* of w if $w^i = u$, for some $i \in \mathbb{N} \cup \{\infty\}$ with $i \leq |w|$.

A (Σ-labeled) *tree* is a function $t : T \to \Sigma$, where $T \subseteq \mathbb{N}^*$ satisfies the following conditions: (i) T is prefix-closed (i.e., if $w \in T$ and u is a prefix of w then $u \in T$) and (ii) if $xi \in T$ and $i > 0$ then $x(i-1) \in T$. The elements in T are called the *nodes* of t and the empty word ε is called the *root* of t. A node $xi \in T$ with $i \in \mathbb{N}$ is called a *child* of the node $x \in T$. An (infinite) *path* in t is a word $\pi \in \mathbb{N}^\omega$ such that $u \in T$, for every prefix u of π. We write $t(\pi)$ for the word $t(\pi^0)t(\pi^1)\dots \in \Sigma^\omega$.

For a set P of propositions, $\mathcal{B}^+(P)$ is the set of *positive Boolean formulas* over P, i.e., the formulas built from the propositions in P, and the connectives $\wedge$ and $\vee$. Given $M \subseteq P$ and $\beta \in \mathcal{B}^+(P)$, we write $M \models \beta$ if the assignment that assigns true to the propositions in M and assigns false to the propositions in $P \setminus M$ satisfies β. Moreover, we write $M \models\!\!\!\!\cdot\; \beta$ if M is a *minimal model* of β, i.e., $M \models \beta$ and there is no $p \in M$ such that $M \setminus \{p\} \models \beta$.

In the following, we define 2-way alternating automata, which scan input words letter by letter with their read-only head. The meaning of "2-way" and "alternating" is best illustrated by the example transition $\delta(p, a) = (q, -1) \vee ((r, 0) \wedge (s, 1))$ of such an automaton, where p, q, r, s are states, a is a letter of the input alphabet, and δ is the transition function. The second coordinate of the tuples $(q, -1), (r, 0), (s, 1)$ specify in which direction the read-only head moves: -1

for left, 0 for not moving, and 1 for right. The transition above can be read as follows. When reading the letter a in state p, the automaton has two choices: (i) It goes to state q and moves the read-only head to the left. In this case, the automaton proceeds scanning the input word from the updated state and position. (ii) Alternatively, it can go to state r *and* to state s, where the read-only head is duplicated: the first copy proceeds scanning the input word from the state r, where the position of the read-only head is not altered; the second copy proceeds scanning the input word from the state s, where the read-only head is moved to the right. Note that the choices (i) and (ii) are given by the minimal models of the example transition $\delta(p, a)$, which is a positive Boolean formula with propositions that are pairs of states and movements of the read-only head.

Let $\mathbb{D} := \{-1, 0, 1\}$ be the set of directions in which the read-only head can move. Formally, a *2-way alternating automaton* $\mathcal{A}$ is a tuple $(Q, \Sigma, \delta, q_\mathrm{I}, \mathcal{F})$, where Q is a finite set of states, Σ is a finite nonempty alphabet, $\delta : Q \times \Sigma \to \mathcal{B}^+(Q \times \mathbb{D})$ is the transition function, $q_\mathrm{I} \in Q$ is the initial state, and $\mathcal{F} \subseteq Q^\omega$ is the acceptance condition. The *size* $|\mathcal{A}|$ of the automaton $\mathcal{A}$ is $|Q|$.

A *configuration* of $\mathcal{A}$ is a pair $(q, i) \in Q \times \mathbb{N}$. Intuitively, q is the current state and i is the position of the read-only head in the input word. A *run* of $\mathcal{A}$ on the word $w \in \Sigma^\omega$ is a tree $r : T \to Q \times \mathbb{N}$ such that $r(\varepsilon) = (q_\mathrm{I}, 0)$ and for each node $x \in T$ with $r(x) = (q, j)$, we have that

$$\big\{(q', j' - j) \in Q \times \mathbb{Z} \,\big|\, r(y) = (q', j'),\ \text{where } y \text{ is a child of } x \text{ in } r\big\} \models \delta(q, w_j)\,.$$

Observe that we require here that the set of labels of the children is a minimal model of the positive Boolean formula $\delta(q, w_j)$. Intuitively, the minimality requirement prevents the automaton from doing unnecessary work in an accepting run. We need this minimality requirement in Section 3.3. A path π in r is *accepting* if $q_0 q_1 \cdots \in \mathcal{F}$, where $r(\pi) = (q_0, i_0)(q_1, i_1) \cdots \in (Q \times \mathbb{N})^\omega$. The run r is *accepting* if every path in r is accepting. The *language* of $\mathcal{A}$ is the set $L(\mathcal{A}) := \{w \in \Sigma^\omega \,|\, \text{there is an accepting run of } \mathcal{A} \text{ on } w\}$.

In the following, we introduce restricted classes of 2-way alternating automata. Let $\mathcal{A} = (Q, \Sigma, \delta, q_\mathrm{I}, \mathcal{F})$ be a 2-way alternating automaton.

Note that we do not have any restriction on the acceptance condition $\mathcal{F}$; it can be any subset of Q^ω. However, since this is often too general, one usually considers automata where the acceptance conditions are specified in a certain finite way—the *type* of an acceptance condition. Commonly used types of acceptance conditions are listed in Table 1. Here, $\inf(\pi)$ is the set of states that occur infinitely often in $\pi \in Q^\omega$ and the integer k is called the *index* of the automaton. If $\mathcal{F}$ is specified by the type τ, we say that $\mathcal{A}$ is a τ automaton. Moreover, if the type of the acceptance condition is clear from the context, we just give the finite description α instead of $\mathcal{F}$. For instance, a Büchi automaton is given as a tuple $(S, \Gamma, \eta, s_\mathrm{I}, \alpha)$ with $\alpha \subseteq S$.

The automaton $\mathcal{A}$ is *1-way* if $\delta(q, a) \in \mathcal{B}^+(Q \times \{1\})$, for all $q \in Q$ and $a \in \Sigma$. That means, $\mathcal{A}$ can only move the read-only head to the right. If $\mathcal{A}$ is 1-way, we assume that δ is of the form $\delta : Q \times \Sigma \to \mathcal{B}^+(Q)$.

The automaton $\mathcal{A}$ is *nondeterministic* if δ returns a disjunction of propositions for all inputs; $\mathcal{A}$ is *universal* if δ returns a conjunction of propositions for

Table 1. Types of acceptance conditions

type: τ	finite description, acceptance condition: α, $\mathcal{F}$
Büchi co-Büchi	$\alpha = F \subseteq Q$ $\mathcal{F} := \{\pi \in Q^\omega \mid \inf(\pi) \cap F \neq \emptyset\}$ $\mathcal{F} := \{\pi \in Q^\omega \mid \inf(\pi) \cap F = \emptyset\}$
parity co-parity	$\alpha = \{F_1, \ldots, F_{2k}\} \subseteq 2^Q$, where $F_1 \subseteq F_2 \subseteq \cdots \subseteq F_{2k}$ $\mathcal{F} := \{\pi \in Q^\omega \mid \min\{i \mid F_i \cap \inf(\pi) \neq \emptyset\} \text{ is even}\}$ $\mathcal{F} := \{\pi \in Q^\omega \mid \min\{i \mid F_i \cap \inf(\pi) \neq \emptyset\} \text{ is odd}\}$
Rabin Streett	$\alpha = \{(B_1, C_1), \ldots, (B_k, C_k)\} \subseteq 2^Q \times 2^Q$ $\mathcal{F} := \bigcup_i \{\pi \in Q^\omega \mid \inf(\pi) \cap B_i \neq \emptyset \text{ and } \inf(\pi) \cap C_i = \emptyset\}$ $\mathcal{F} := \bigcap_i \{\pi \in Q^\omega \mid \inf(\pi) \cap B_i = \emptyset \text{ or } \inf(\pi) \cap C_i \neq \emptyset\}$
Muller	$\alpha = \{M_1, \ldots, M_k\} \subseteq 2^Q$ $\mathcal{F} := \bigcup_i \{\pi \in Q^\omega \mid \inf(\pi) = M_i\}$

all inputs; $\mathcal{A}$ is *deterministic* if it is nondeterministic and universal. For nondeterministic and deterministic automata, we use standard notation. For instance, if $\mathcal{A}$ is nondeterministic, we view δ as a function of the form $\delta : Q \to 2^{Q \times \mathbb{D}}$. That means, a clause is written as a set. Note that a run $r : T \to Q \times \mathbb{N}$ of a nondeterministic automaton $\mathcal{A}$ on $w \in \Sigma^\omega$ consists of the single path $\pi = 0^\omega$. To increase readabiliy, we call $r(\pi) \in (Q \times \mathbb{N})^\omega$ also a run of $\mathcal{A}$ on w. Moreover, for $R \subseteq Q$ and $a \in \Sigma$, we abbreviate $\bigcup_{q \in R} \delta(q, a)$ by $\delta(R, a)$.

3 Alternation-Elimination Scheme

In this section, we present a general construction scheme for translating alternating automata into language-equivalent nondeterministic automata. The construction scheme is general in the sense that it can be instantiated for different classes of alternating automata. We provide such instances in Section 4. Before presenting the construction scheme in Section 3.2, we need some preparatory work, which we present in Section 3.1.

3.1 Memoryless Runs as Words

Let $\mathcal{A} = (Q, \Sigma, \delta, q_I, \mathcal{F})$ be a 2-way alternating automaton and let $r : T \to Q \times \mathbb{N}$ be a run of $\mathcal{A}$ on the word $w \in \Sigma^\omega$. The run r is *memoryless*[1] if all equally labeled nodes $x, y \in T$ have isomorphic subtrees, i.e., if $r(x) = r(y)$ then for all $z \in \mathbb{N}^*$, $xz \in T \Leftrightarrow yz \in T$ and whenever $xz \in T$ then $r(xz) = r(yz)$. We define

$$M(\mathcal{A}) := \{w \in \Sigma^\omega \mid \text{there is an accepting memoryless run on } w\}.$$

[1] The choice of the term "memoryless" becomes clear when viewing a run of an alternating automaton as a representation of a strategy of the first player in a two-person infinite game [24]. A memoryless run encodes a memoryless strategy (also known as a positional strategy) of the first player, i.e., a strategy that does not take the history of a play into account.

Note that $M(\mathcal{A}) \subseteq L(\mathcal{A})$; however, the converse does not hold in general. The languages are equal when $\mathcal{A}$ is an alternating Büchi, co-Büchi, or parity automaton [8, 17], or an alternating Rabin automaton [14]. For alternating Streett and Muller automata, the languages can be different. However, such automata can be translated to language-equivalent alternating parity automata, see [20].

Observe that in a memoryless run $r : T \to Q \times \mathbb{N}$ of $\mathcal{A}$ on a word $w \in \Sigma^\omega$, we can merge nodes with isomorphic subtrees without loosing any information. We obtain an infinite directed graph, which can be represented as an infinite word of functions $f \in (Q \to 2^{Q \times \mathbb{D}})^\omega$, where $f_j(q)$ returns the labels of the children of a node $x \in T$ with label (q, j). Note that $f_j(q)$ is well-defined, since $x \in T$ and $y \in T$ have isomorphic subtrees whenever $r(x) = r(y)$.

Definition 1. *The* induced tree[2] *$t : T \to Q \times \mathbb{N}$ of the word $f \in (Q \to 2^{Q \times \mathbb{D}})^\omega$ is inductively defined:*

(i) we have that $\varepsilon \in T$ and $t(\varepsilon) := (q_\mathrm{I}, 0)$, and
(ii) for each $x \in T$ with $t(x) = (p, j)$ and $f_j(p) = \{(q_0, d_0), \ldots, (q_k, d_k)\}$, we have that $x0, \ldots, xk \in T$ and $t(xi) := (q_i, j + d_i)$, for each $i \in \mathbb{N}$ with $i \leq k$.

The word $f \in (Q \to 2^{Q \times \mathbb{D}})^\omega$ is a run-word *of $\mathcal{A}$ on $w \in \Sigma^\omega$ if the induced tree t is a run of $\mathcal{A}$ on w. Moreover, f is* accepting *if t is accepting. Finally, we define $L'(\mathcal{A}) := \{w \in \Sigma^\omega \mid$ there is an accepting run-word of $\mathcal{A}$ on $w\}$.*

The following lemma states that automata that accept by run-words are as expressive as automata that accept by memoryless runs.

Lemma 2. *For every 2-way alternating automaton $\mathcal{A}$, $M(\mathcal{A}) = L'(\mathcal{A})$.*

3.2 Reduction to Complementation

For the following, fix a 2-way alternating automaton $\mathcal{A} = (Q, \Sigma, \delta, q_\mathrm{I}, \mathcal{F})$. Moreover, we abbreviate the function space $Q \to 2^{Q \times \mathbb{D}}$ by Γ. Without loss of generality, we assume that $\mathcal{A}$ has a rejecting sink state $s \in Q$, i.e., for every $a \in \Sigma$, $\delta(s, a) = (s, 1)$ and for every $u \in Q^*$, $us^\omega \notin \mathcal{F}$.

From $\mathcal{A}$ we construct the automaton $\mathcal{B} := (Q, \Sigma \times \Gamma, \eta, q_\mathrm{I}, Q^\omega \setminus \mathcal{F})$, which is 2-way and nondeterministic. For $q \in Q$ and $(a, g) \in \Sigma \times \Gamma$, we define the transition function as

$$\eta\big(q, (a, g)\big) := \begin{cases} g(q) & \text{if } g(q) \models \delta(q, a), \\ \{(s, 1)\} & \text{otherwise.} \end{cases}$$

The next lemma is at the core of the results of this paper. It states that the automaton $\mathcal{B}$ rejects exactly those words $(w_0, f_0)(w_1, f_1) \cdots \in (\Sigma \times \Gamma)^\omega$, where f is an accepting run-word of $\mathcal{A}$ on w.

[2] The tree t is actually not uniquely determined, since we do not uniquely order the children of a node in t. However, the order of the children is irrelevant for the results in this paper, and to simplify matters, we consider two trees as isomorphic if they only differ in the order of their subtrees.

Lemma 3. *For all words $w \in \Sigma^\omega$ and $f \in \Gamma^\omega$, it holds that*

$$(w_0, f_0)(w_1, f_1)\cdots \in L(\mathcal{B}) \quad \text{iff} \quad \begin{array}{l}\text{(i) } f \text{ is not a run-word of } \mathcal{A} \text{ on } w, \text{ or}\\ \text{(ii) } f \text{ is a rejecting run-word of } \mathcal{A} \text{ on } w.\end{array}$$

The next theorem shows that when $L(\mathcal{A}) = M(\mathcal{A})$, the problem of eliminating the alternation of $\mathcal{A}$ (i.e., to construct a language-equivalent nondeterministic automaton) reduces to the problem of complementing the nondeterministic automaton $\mathcal{B}$.

Theorem 4. *Let $\mathcal{C}$ be a nondeterministic automaton that accepts the complement of $L(\mathcal{B})$ and let $\mathcal{D}$ be the projection of $\mathcal{C}$ on Σ. If $L(\mathcal{A}) = M(\mathcal{A})$ then $L(\mathcal{A}) = L(\mathcal{D})$.*

Proof. For a word $w \in \Sigma^\omega$, the following equivalences hold:

$$\begin{array}{ll} & w \in L(\mathcal{A})\\ \overset{\text{Lemma 2}}{\Leftrightarrow} & w \in L'(\mathcal{A})\\ \overset{\text{Lemma 3}}{\Leftrightarrow} & (w_0, f_0)(w_1, f_1)\cdots \notin L(\mathcal{B}), \text{ for some } f \in \Gamma^\omega\\ \Leftrightarrow & (w_0, f_0)(w_1, f_1)\cdots \in L(\mathcal{C}), \text{ for some } f \in \Gamma^\omega\\ \Leftrightarrow & w \in L(\mathcal{D})\,. \end{array}$$

□

3.3 On Weak and Loop-Free Automata

In the following, we show that the nondeterministic automaton $\mathcal{B}$ inherits properties from the alternating automaton $\mathcal{A}$. We exploit these properties in Section 4.

Weak Automata. The notion of weakness for automata was introduced in [23]. It led to new insights (e.g., [6, 16, 17, 23]). Moreover, many operations on weak automata are often simpler and more efficient to implement than their counterparts for non-weak automata, see e.g., [11, 17].

The following definition of weakness for an arbitrary acceptance condition $\mathcal{F}$ generalizes the standard definition of weakness for the Büchi acceptance condition. Let $\mathcal{A} = (Q, \Sigma, \delta, q_\mathrm{I}, \mathcal{F})$ be a 2-way alternating automaton. A set of states $S \subseteq Q$ is *accepting* if $\inf(r(\pi)) \subseteq S$ implies $r(\pi) \in \mathcal{F}$, for each run r and each path π in r. S is *rejecting* if $\inf(r(\pi)) \subseteq S$ implies $r(\pi) \notin \mathcal{F}$, for each run r and each path π in r. The automaton $\mathcal{A}$ is (inherently) *weak*, if there is a partition on Q into the sets $Q_1, \ldots, Q_n$ such that (i) each Q_i is either accepting or rejecting, and (ii) there is a partial order $\preceq$ on the Q_is such that for every $p \in Q_i$, $q \in Q_j$, $a \in \Sigma$, and $d \in \mathbb{D}$: if (q, d) occurs in $\delta(p, a)$ then $Q_j \preceq Q_i$. The automaton $\mathcal{A}$ is *very-weak* (also known as 1-weak or linear), if each Q_i is a singleton. The intuition of weakness is that each path of any run of a weak automaton that gets trapped in one of the Q_is is accepting iff Q_i is accepting.

The following lemma shows that in our alternation-elimination scheme, the weakness of an alternating automaton $\mathcal{A}$ transfers to the nondeterministic automaton $\mathcal{B}$, which needs to be complemented (see Theorem 4).

Lemma 5. *Let $\mathcal{A}$ be a 2-way alternating automaton and let $\mathcal{B}$ be the 2-way nondeterministic automaton as defined in Section 3.2. If $\mathcal{A}$ is weak then $\mathcal{B}$ is weak, and if $\mathcal{A}$ is very-weak then $\mathcal{B}$ is very-weak.*

Table 2. Sizes of 1-way NBAs obtained by instances of the construction scheme

		VABA	ABA	APA	ARA
1-way	size	$O(n2^n)$	$O(2^{2n})$	$2^{O(nk \log n)}$	$2^{O(nk \log nk)}$
	compl.	by Corollary 11	by [4]	by [18]	by [18]
2-way	size	$O(n2^{3n})$	$2^{O(n^2)}$	$2^{O((nk)^2)}$	
	compl.	by Theorem 10	by [28]	by [28]	
2-way +	size	$O(n2^{2n})$	$O(2^{4n})$		
loop-free	compl.	by Corollary 9	by Theorem 8		

Loop-Free Automata. For a 2-way alternating automaton $\mathcal{A} = (Q, \Sigma, \delta, q_I, \mathcal{F})$, we define the set $\Pi(\mathcal{A})$ as the set of words $(q_0, j_0)(q_1, j_1) \cdots \in (Q \times \mathbb{N})^\omega$ such that $(q_0, j_0) = (q_I, 0)$ and for all $i \in \mathbb{N}$, there is some $a \in \Sigma$ and a minimal model $M \subseteq Q$ of $\delta(q_i, a)$ with $(q_{i+1}, j_{i+1} - j_i) \in M$. The automaton $\mathcal{A}$ is *loop-free* if for all words $\pi \in \Pi(\mathcal{A})$, there are no integers $i, j \in \mathbb{N}$ with $i \neq j$ such that $\pi_i = \pi_j$. Recall that π_i and π_j are configurations, which consist of the current state and the position of the read-only head. So, $\mathcal{A}$ does not loop on a branch in a partial run when scanning an input word.

As in the case of weak automata, the nondeterministic automaton $\mathcal{B}$ inherits the loop freeness of the alternating automaton $\mathcal{A}$ in the construction scheme.

Lemma 6. *Let $\mathcal{A}$ be a 2-way alternating automaton and let $\mathcal{B}$ be the 2-way nondeterministic automaton as defined in Section 3.2. If $\mathcal{A}$ is loop-free then $\mathcal{B}$ is loop-free.*

4 Instances of the Alternation-Elimination Scheme

In this section, we give instances of the construction scheme presented in Section 3. For brevity, we use the following acronyms: ABA for *1-way alternating Büchi automaton*, NBA for *1-way nondeterministic Büchi automaton*, and DBA for *1-way deterministic Büchi automaton*. We prepend the symbols 2, W, and V to the acronyms to denote 2-way, weak, and very-weak automata, respectively. Analogously, we use acronyms for co-Büchi, parity, Rabin, and Streett automata. For instance, co-2WNBA abbreviates *2-way weak nondeterministic co-Büchi automaton*.

Table 2 summarizes some instances of our construction scheme for obtaining language-equivalent NBAs from alternating automata. The table states the sizes of the resulting NBAs, where n is the size and k is the index of the given alternating automaton. Moreover, for each instance in Table 2, we reference the used complementation construction. We remark that the classes of alternating automata in the columns VABA, ABA, and APA in Table 2 are relevant in finite-state model checking, since system properties that are given as formulas of the widely used temporal logics like LTL, PSL, and μLTL or fragments thereof can directly be translated into alternating automata that belong to one of these classes of automata.

All the instances in Table 2 follow the same pattern, which is as follows. Let us use the notation from Section 3.2. In particular, $\mathcal{A}$ is the given alternating automaton over the alphabet Σ for which we want to construct a language-equivalent NBA $\mathcal{D}$.

1. From $\mathcal{A}$ we construct the nondeterministic automaton $\mathcal{B}$ over the extended alphabet $\Sigma \times \Gamma$ with the co-acceptance condition of $\mathcal{A}$.
2. We complement the nondeterministic automaton $\mathcal{B}$ with the complementation construction that is referenced in Table 2. We obtain an NBA $\mathcal{C}$ over the alphabet $\Sigma \times \Gamma$. Note that in some instances it is necessary to switch to another acceptance condition in order to apply the referenced complementation construction. In these cases, we first transform $\mathcal{B}$ accordingly. Such transformations are given in [20].
3. Finally, we project the extended alphabet $\Sigma \times \Gamma$ of the NBA $\mathcal{C}$ to Σ. This gives us the NBA $\mathcal{D}$.

For instance, if $\mathcal{A}$ is an ARA, we construct an NSA $\mathcal{B}$. With the construction from [18], we complement the NSA $\mathcal{B}$ and obtain an NBA $\mathcal{C}$.

Note that with the construction scheme at hand, the only remaining difficult part is the complementation construction in the second step. In the following Section 4.1, we present novel complementation constructions that are used by some of the instances of the construction scheme from Table 2.

4.1 Novel Complementation Constructions

Complementing Loop-Free co-2NBAs. The following construction can be seen as a combination of Vardi's complementation construction [29] for 2-way nondeterministic automata over finite words and the Miyano-Hayashi construction [17, 22] for 1-way alternating Büchi automata. The construction is based on the following characterization of the words that are rejected by a loop-free co-2NBA.

Lemma 7. *Let $\mathcal{A} = (Q, \Sigma, \delta, q_{\mathrm{I}}, F)$ be a loop-free co-2NBA and $w \in \Sigma^\omega$. It holds that $w \notin L(\mathcal{A})$ iff there are words $R \in (2^Q)^\omega$ and $S \in (2^{Q \setminus F})^\omega$ such that*

(i) $q_{\mathrm{I}} \in R_0$,
(ii) *for all* $i \in \mathbb{N}$, $p, q \in Q$, *and* $d \in \mathbb{D}$, *if* $p \in R_i$, $(q, d) \in \delta(p, w_i)$, *and* $i + d \geq 0$ *then* $q \in R_{i+d}$,
(iii) $S_0 = R_0 \setminus F$,
(iv) *for all* $i \in \mathbb{N}$, $p, q \in Q \setminus F$, *and* $d \in \{0, 1\}$, *if* $p \in S_i$ *and* $(q, d) \in \delta(p, w_i)$ *then* $q \in S_{i+d}$,
(v) *for all* $i \in \mathbb{N}$ *and* $p, q \in Q \setminus F$, *if* $p \in S_i$, $(q, -1) \in \delta(p, w_i)$, *and* $i - 1 \geq 0$ *then* $q \in S_{i-1}$ *or* $S_{i-1} = \emptyset$, *and*
(vi) *there are infinitely many* $i \in \mathbb{N}$ *such that* $S_i = \emptyset$ *and* $S_{i+1} = R_{i+1} \setminus F$.

Proof (sketch). ($\Rightarrow$) Assume $w \notin L(\mathcal{A})$, i.e., all runs of $\mathcal{A}$ on w visit a state in F infinitely often. We need the following definitions. A word $(q_0, j_0) \dots (q_n, j_n) \in (Q \times \mathbb{N})^*$ is a *run segment* if $(q_{i+1}, j_{i+1} - j_i) \in \delta(q_i, w_i)$, for all $i < n$. The

run segment is *initial* if $(q_0, j_0) = (q_\mathrm{I}, 0)$. For $i \in \mathbb{N}$, we define $R_i := \{q_n \in Q \mid \text{there is an initial run segment } (q_0, j_0) \dots (q_n, j_n) \text{ with } j_n = i\}$. Since $(q_\mathrm{I}, 0)$ is an initial run segment, R satisfies (i). To show that (ii) holds, assume $i \in \mathbb{N}$, $p, q \in Q$, and $d \in \mathbb{D}$. If $p \in R_i$, $(q, d) \in \delta(p, w_i)$, and $i + d \geq 0$ then there is an initial run segment $r_0 \dots r_n \in (Q \times \mathbb{N})^*$ such that $r_n = (p, i)$. Hence, $r_0 \dots r_n(q, i+d) \in (Q \times \mathbb{N})^*$ is also an initial run segment and therefore, $q \in R_{i+d}$.

It remains to define $S \in (2^{Q \setminus F})^\omega$ that satisfies (iii)–(vi). In the following, we call a run segment $(q_0, j_0) \dots (q_n, j_n) \in (Q \times \mathbb{N})^*$ *F-avoiding* if $q_i \notin F$, for all $i \leq n$. For defining S inductively, it is convenient to use the auxiliary set $S_{-1} := \emptyset$.

Let $m \in \mathbb{N} \cup \{-1\}$ such that $S_m = \emptyset$. We define $T \in (Q \times \mathbb{N})^\omega$ as the set of F-avoiding run segments that start in $R_{m+1} \setminus F$, i.e., $T_i := \{q_k \in Q \mid \text{there is an } F\text{-avoiding run segment } (q_0, j_0) \dots (q_k, j_k) \text{ with } q_0 \in R_{m+1}, j_0 = m+1, \text{ and } j_k = i\}$, for $i \in \mathbb{N}$. We show that there is an integer $n \in \mathbb{N}$ such that $T_n = \emptyset$. Assume that such an integer n does not exist. With König's Lemma it is easy to see that T contains an infinite F-avoiding run segment. Thus, there is an accepting infinite run of $\mathcal{A}$ on w. This contradicts the assumption $w \notin L(\mathcal{A})$. We choose $n \in \mathbb{N}$ to be minimal and define $S_{m+1+i} := T_i$, for $i \leq n$.

By construction of S, conditions (iii) and (vi) are satisfied. With a similar argumentation that we used to show (ii), we see that (iv)–(v) hold.

($\Leftarrow$) Assume there are words $R \in (2^Q)^\omega$ and $S \in (2^{Q \setminus F})^\omega$ with the conditions (i)–(vi). Let $r := (q_0, j_0)(q_1, j_1) \cdots \in (Q \times \mathbb{N})^\omega$ be a run of $\mathcal{A}$ on w. Due to conditions (i) and (ii), we have $q_i \in R_{j_i}$, for each $i \in \mathbb{N}$. We show that r is rejecting.

Suppose that r is accepting. There is a $k \in \mathbb{N}$ such that $q_i \notin F$, for all $i > k$. Due to condition (vi), there is a breakpoint $S_m = \emptyset$ with $m > j_k$ and $S_{m+1} = R_{m+1} \setminus F$. Since r is loop-free, there is an $h > k$ such that $j_h = m + 1$. Without loss of generality, we assume that h is maximal. Since r is loop-free and the set Q is finite, such an h exists. We have $j_i > m + 1$, for all $i > h$.

Since $q_h \in R_{j_h}$ and $q_h \notin F$, we have $q_h \in S_{j_h}$. Using the conditions (iv) and (v), we obtain by induction that $q_i \in S_{j_i}$, for all $i > h$. Since r is loop-free, there is no $n > m$ such that $S_n = \emptyset$. We obtain a contradiction to condition (vi). □

The following theorem extends the Miyano-Hayashi construction to 2-way automata. Roughly speaking, the constructed NBA $\mathcal{C}$ guesses a run that satisfies the conditions of Lemma 7, for the given co-2NBA and an input word.

Theorem 8. *For a loop-free co-2NBA $\mathcal{B}$, there is an NBA $\mathcal{C}$ that accepts the complement of $L(\mathcal{B})$ and has $1 + 2^{4|\mathcal{B}|}$ states.*

Proof (sketch). Let $\mathcal{B} = (Q, \Sigma, \delta, q_\mathrm{I}, F)$ be a loop-free co-2NBA. We define the NBA $\mathcal{C} := (P, \Sigma, \eta, p_\mathrm{I}, G)$, where $P := (2^Q \times 2^{Q \setminus F} \times 2^Q \times 2^{Q \setminus F}) \cup \{p_\mathrm{I}\}$, and $G := 2^Q \times \{\emptyset\} \times 2^Q \times 2^{Q \setminus F}$. The transition function η is defined as follows. For the initial state p_I and $a \in \Sigma$, we have that $\eta(p_\mathrm{I}, a) \ni (R_0, S_0, R_1, S_1)$ iff the following conditions hold:

- $q_\mathrm{I} \in R_0$,
- for all $p \in R_0$, $q \in Q$, and $d \in \{0, 1\}$: if $(q, d) \in \delta(p, a)$ then $q \in R_d$,

– $S_0 = R_0 \setminus F$,
– for all $p \in S_0$, $q \notin F$, and $d \in \{0, 1\}$, if $(q, d) \in \delta(p, a)$ then $q \in S_d$.

For the other states in P and $a \in \Sigma$, we have that $\eta\big((R_{-1}, S_{-1}, R_0, S_0), a\big) \ni (R_0, S_0, R_1, S_1)$ iff the following conditions hold:

– for all $p \in R_0$, $q \in Q$, and $d \in \mathbb{D}$, if $(q, d) \in \delta(p, a)$ then $q \in R_d$,
– for all $p \in S_0$, $q \notin F$, and $d \in \{0, 1\}$, if $(q, d) \in \delta(p, a)$ then $q \in S_d$,
– for all $p \in S_0$ and $q \notin F$, if $(q, -1) \in \delta(p, a)$ then $q \in S_{-1}$ or $S_{-1} = \emptyset$, and
– if $S_0 = \emptyset$ then $S_1 = R_1 \setminus F$.

Intuitively, for an input word w, the automaton guesses the words $R \in (2^Q)^\omega$ and $S \in (2^{Q \setminus F})^\omega$ from Lemma 7. With the first and third component of P, it checks the conditions (i) and (ii). With the second and last component, it checks that (iii)–(v) holds. Finally, the acceptance condition ensures that (vi) is satisfied. It easy to check that $\mathcal{C}$ accepts the complement of $L(\mathcal{B})$. □

Complementing (Loop-Free) co-2VNBA. If the given automaton is a loop-free co-2VNBA, we can simplify the 2-way breakpoint construction presented in Theorem 8. The simplification is based on the following observation: each run of a very-weak automaton will eventually get trapped in a state with a self-loop. Thus, the conditions (iii)–(vi) from Lemma 7 can be simplified accordingly. The simpler conditions allow us to optimize the complementation construction for loop-free co-2VNBA. Roughly speaking, instead of guessing the word $S \in (2^{Q \setminus F})^\omega$ from Lemma 7 and checking that S fulfills the conditions (iii)–(vi), the constructed automaton only has to check that no run of the loop-free co-2VNBA gets trapped in a state $q \notin F$.

Additionally, for very-weak automata, we can extend the above 2-way breakpoint construction so that it can deal with non-loop-free co-2VNBAs. This extension is based on the observation that there are only two types of loops: a very-weak automaton loops if (1) it gets trapped in a state without moving the read-only head or (2) it gets trapped in a state in which it first moves the read-only head to the right and then to the left. Such loops can be locally detected.

Based on these two observations, we obtain from Lemma 7 the following corollary that characterizes the words that are rejected by a given (loop-free) co-2VNBA. We exploit this new characterization in the Theorem 10 below for complementing (loop-free) co-2VNBAs.

Corollary 9. *Let $\mathcal{A} = (Q, \Sigma, \delta, q_I, F)$ be a co-2VNBA and $w \in \Sigma^\omega$. It holds that $w \notin L(\mathcal{A})$ iff there is a word $R \in (2^Q)^\omega$ such that*

(i) $q_I \in R_0$,
(ii) *for all $i \in \mathbb{N}$, $p, q \in Q$, and $d \in \mathbb{D}$, if $p \in R_i$, $(q, d) \in \delta(p, w_i)$, and $i + d \geq 0$ then $q \in R_{i+d}$,*
(iii) *there is no $n \in \mathbb{N}$ such that $q \in R_i \setminus F$ and $(q, 1) \in \delta(q, w_i)$, for all $i \geq n$.*
(iv) *there is no $i \in \mathbb{N}$ and $q \in R_i \setminus F$ such that $(q, 0) \in \delta(q, w_i)$, and*

(v) there is no $i \in \mathbb{N}$ *and* $q \in R_i \setminus F$ *such that* $(q, 1) \in \delta(q, w_i)$ *and* $(q, -1) \in \delta(q, w_{i+1})$.

Furthermore, when $\mathcal{A}$ *is loop-free only (i)–(iii) must hold.*

Theorem 10. *For a co-2VNBA* $\mathcal{B}$*, there is an NBA* $\mathcal{C}$ *that accepts the complement of* $L(\mathcal{B})$ *and has* $O(|\mathcal{B}| \cdot 2^{3|\mathcal{B}|})$ *states. If* $\mathcal{B}$ *is loop-free then we can construct* $\mathcal{C}$ *with* $O(|\mathcal{B}| \cdot 2^{2|\mathcal{B}|})$ *states.*

Proof (sketch). Let $\mathcal{B} = (Q, \Sigma, \delta, q_{\mathrm{I}}, F)$ be a co-2VNBA, where we assume that $Q = \{1, \ldots, n\}$ and $Q \setminus F = \{1, \ldots, m\}$, for $m, n \in \mathbb{N}$. If $m = 0$ then $F = Q$ and hence, $L(\mathcal{B}) = \emptyset$. So, assume $m > 0$. Furthermore, assume $m < n$. Otherwise, we add an additional accepting state to Q. Let $k := m + 1$.

We define the NBA $\mathcal{C} := (P, \Sigma, \eta, p_{\mathrm{I}}, G)$, where $P := (2^Q \times 2^Q \times 2^{Q \setminus F} \times \{1, \ldots, k\}) \cup \{p_{\mathrm{I}}\}$ and $G := 2^Q \times 2^Q \times 2^{Q \setminus F} \times \{1\}$. The transition function η is defined as follows. For the initial state p_{I} and $a \in \Sigma$, we have that $\eta(p_{\mathrm{I}}, a) \ni (R_0, R_1, R'_0, 1)$ iff the following conditions hold:

- $q_{\mathrm{I}} \in R_0$,
- for all $p \in R_0$, $q \in Q$, and $d \in \{0, 1\}$, if $(q, d) \in \delta(p, a)$ then $q \in R_d$,
- there is no $q \in R_0 \setminus F$ such that $(q, 0) \in \delta(q, a)$, and
- $R'_0 = \{q \in R_0 \mid (q, 1) \in \delta(q, a)\}$.

For the other states in P and $a \in \Sigma$, we have that $\eta\big((R_{-1}, R_0, R'_{-1}, s), a\big) \ni (R_0, R_1, R'_0, s')$ iff the following conditions hold:

- for all $p \in R_0$, $q \in Q$, and $d \in \mathbb{D}$, if $(q, d) \in \delta(p, a)$ then $q \in R_d$,
- $s' = s$, if $s \leq m$, $s \in R_0$, and $(s, 1) \in \delta(s, a)$; otherwise, $s' = (s \bmod k) + 1$,
- there is no $q \in R_0 \setminus F$ such that $(q, 0) \in \delta(q, a)$,
- there is no $q \in R'_{-1}$ such that $(q, -1) \in \delta(q, a)$, and
- $R'_0 = \{q \in R_0 \mid (q, 1) \in \delta(q, a)\}$.

It remains to show that $\mathcal{C}$ accepts the complement of $L(\mathcal{B})$. Note that $\mathcal{C}$ locally checks all conditions of Corollary 9 except for (iii). Condition (iii) is satisfied if the run is accepting.

We remark that we need the third component in a state because $\mathcal{C}$ forgets the previously read letter. There is an alternative construction, namely, we construct an automaton with the state space $(2^Q \times 2^Q \times \Sigma \times (Q \setminus F)) \cup \{p_{\mathrm{I}}\}$ that stores the letter in the third component of a state.

When $\mathcal{B}$ is loop-free, the automaton $\mathcal{C}$ does not have to check (iv) and (v). Hence we can drop the third component in $\mathcal{C}$'s state space. □

If the given automaton $\mathcal{A}$ is a co-VNBA, we can further simplify the construction. To ensure that a word w is rejected by the co-VNBA $\mathcal{A}$, one only has to check the first three conditions of Corollary 9, where we can restrict d to 1 instead of $d \in \mathbb{D}$ in condition (ii). We point out that the idea of this construction is implicitly used in the translation [3, 11] of VABAs to NBAs and in the "focus approach" of the satisfiability checking of LTL formulas in [19].

Corollary 11. *For a co-VNBA $\mathcal{B}$, there is a DBA $\mathcal{C}$ that accepts the complement of $L(\mathcal{B})$ and has $O(|\mathcal{B}| \cdot 2^{|\mathcal{B}|})$ states.*

Proof (sketch). Let $\mathcal{B} = (Q, \Sigma, \delta, q_{\mathrm{I}}, F)$. Assume that $Q = \{1, \ldots, n\}$ and $Q \setminus F = \{1, \ldots, m\}$, for $m, n \in \mathbb{N}$. If $m = 0$ then $F = Q$ and hence, $L(\mathcal{B}) = \emptyset$. So, assume $m > 0$. Furthermore, assume that $m < n$. Otherwise, we add an additional accepting state to Q. Let $k := m + 1$. We define the DBA $\mathcal{C} := (2^Q \times \{1, \ldots, k\}, \Sigma, \eta, (\{q_{\mathrm{I}}\}, 1), 2^Q \times \{1\})$, where

$$\eta\big((R, s), a\big) := \begin{cases} \big(\delta(R, a), s\big) & \text{if } s \leq m,\ s \in R, \text{ and } s \in \delta(q, a), \\ \big(\delta(R, a), (s \bmod k) + 1\big) & \text{otherwise.} \end{cases}$$

$\mathcal{B}$ accepts a word w iff there is a run that gets trapped in a state $q \notin F$ iff $\mathcal{C}$ detects the existence of such a run with its second component and rejects. □

4.2 Revisiting Alternation-Elimination Constructions

Let us first review the construction of the nondeterministic automaton $\mathcal{B}$ according to the construction scheme in Section 3.2. Observe that $\mathcal{B}$ possesses the alphabet $\Sigma \times \Gamma$, which is exponential in the size of the given alternating automaton $\mathcal{A}$. In practice, it will not be feasible to explicitly construct $\mathcal{B}$. Fortunately, for the instances in Table 2, we can optimize the constructions by merging the steps of constructing $\mathcal{B}$ and complementing $\mathcal{B}$: we build the transitions of $\mathcal{B}$ only locally and we directly project the extended alphabet $\Sigma \times \Gamma$ to Σ when constructing the complement automaton of $\mathcal{B}$.

For the remainder of this section, let us revisit previously proposed alternation-elimination constructions. The alternation-elimination constructions in [7, 15, 28, 30] for specific classes of alternating automata have a similar flavor as the instances that we obtain from the construction scheme presented in Section 3. In fact, at the core of all these constructions is the complementation of a nondeterministic automaton $\mathcal{B}$ that processes inputs of the given alternating automaton $\mathcal{A}$ augmented with additional information about the runs of the automaton $\mathcal{A}$. However, the previously proposed constructions and the corresponding instances from our construction scheme differ in the following technical detail. The constructions in [7, 15, 28, 30] use an additional automaton $\mathcal{B}'$ that checks whether such an augmented input is valid, i.e., in our terminology that the additional information is a run-word. In the worst-case, the size of $\mathcal{B}'$ is exponential in the size of $\mathcal{A}$. We do not need this additional automaton $\mathcal{B}'$. Instead, the requirement in our construction scheme that the given alternating automaton $\mathcal{A}$ has a rejecting sink state takes care of invalid inputs. This technical detail leads to slightly better upper bounds on the size of the constructed nondeterministic automata, since we do not need to apply the product construction with the automaton $\mathcal{B}'$ to check whether an input is valid.

Finally, we remark that the alternation-elimination construction by Miyano and Hayashi [22] for ABAs, and the constructions by Gastin and Oddoux [11, 12]

for VABAs and loop-free 2VABAs coincide (modulo some minor technical details) with the corresponding instances that we obtain from the presented construction scheme. Moreover, these three alternation-elimination constructions can be seen as special cases of the alternation-elimination construction for loop-free 2ABAs that we obtain from the construction scheme by using the complementation construction in Theorem 8. We are not aware of any other alternation-elimination construction for this class of automata except the one that also handles non-loop-free ABAs. However, the upper bounds for the construction for 2ABAs is worse than the upper bound that we obtain by this new construction for loop-free 2ABAs (see Table 2).

5 Conclusion

We have presented a general construction scheme for translating alternating automata into language-equivalent nondeterministic automata. Furthermore, we have given instances of this construction scheme for various classes of alternating automata. Some of these instances clarify, simplify, or improve existing ones; some of these instances are novel. Since declarative specification languages for reactive systems like LTL or fragments of PSL can directly be translated into some of the considered classes of alternating automata, the presented constructions are of immediate practical interest in finite-state model checking and satisfiability checking.

We remark that the presented constructions depend on complementation constructions for nondeterministic automata. Improving the latter ones, will immediately result in better alternation-elimination constructions. A comparison of the upper bounds on the sizes of the produced nondeterministic automata suggests that alternation elimination for 2-way alternating automata causes a slightly larger blow-up than for 1-way alternating automata (see Table 2). It remains as future work to close this gap, e.g., by providing worst-case examples that match these upper bounds or by improving the constructions.

Acknowledgments. The authors thank Martin Lange, Nir Piterman, and Moshe Vardi for helpful discussions and comments.

References

1. IEEE standard for property specification language (PSL). IEEE Std. 1850TM (October 2005)
2. Banieqbal, B., Barringer, H.: Temporal logic with fixed points. In: Banieqbal, B., Pnueli, A., Barringer, H. (eds.) Temporal Logic in Specification. LNCS, vol. 398. Springer, Heidelberg (1989)
3. Bloem, R., Cimatti, A., Pill, I., Roveri, M.: Symbolic implementation of alternating automata. Int. J. Found. Comput. Sci. 18 (2007)
4. Boigelot, B., Jodogne, S., Wolper, P.: An effective decision procedure for linear arithmetic over the integers and reals. ACM Trans. Comput. Log. 6 (2005)

5. Bustan, D., Fisman, D., Havlicek, J.: Automata construction for PSL, tech. report, Computer Science and Applied Mathematics, The Weizmann Institute of Science, Israel (2005)
6. Chang, E., Manna, Z., Pnueli, A.: The safety-progress classification. In: Logic and Algebra of Specifications. NATO ASI Series (1993)
7. Dax, C., Hofmann, M., Lange, M.: A proof system for the linear time μ-calculus. In: Proc. of FSTTCS 2006. LNCS, vol. 4437 (2006)
8. Emerson, E., Jutla, C.: Tree automata, mu-calculus and determinacy (extended abstract). In: Proc. of FOCS 1991 (1991)
9. Etessami, K., Wilke, T., Schuller, R.: Fair simulation relations, parity games, and state space reduction for Büchi automata. SIAM J. Comput. 34 (2005)
10. Fritz, C., Wilke, T.: Simulation relations for alternating Büchi automata. Theoret. Comput. Sci. 338 (2005)
11. Gastin, P., Oddoux, D.: Fast LTL to Büchi automata translation. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, p. 53. Springer, Heidelberg (2001)
12. Gastin, P., Oddoux, D.: LTL with past and two-way very-weak alternating automata. In: Rovan, B., Vojtáš, P. (eds.) MFCS 2003. LNCS, vol. 2747, pp. 439–448. Springer, Heidelberg (2003)
13. Gerth, R., Peled, D., Vardi, M., Wolper, P.: Simple on-the-fly automatic verification of linear temporal logic. In: Proc. of PSTV 1995. IFIP Conf. Proc., vol. 38 (1996)
14. Jutla, C.: Determinization and memoryless winning strategies. Inf. Comput. 133 (1997)
15. Kupferman, O., Piterman, N., Vardi, M.: Extended temporal logic revisited. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154, p. 519. Springer, Heidelberg (2001)
16. Kupferman, O., Vardi, M.: Weak alternating automata and tree automata emptiness. In: Proc. of STOC 1998 (1998)
17. Kupferman, O., Vardi, M.: Weak alternating automata are not that weak. ACM Trans. Comput. Log. 2 (2001)
18. Kupferman, O., Vardi, M.: Complementation constructions for nondeterministic automata on infinite words. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 206–221. Springer, Heidelberg (2005)
19. Lange, M., Stirling, C.: Focus games for satisfiability and completeness of temporal logic. In: Proc. of LICS 2001 (2001)
20. Löding, C.: Methods for the transformation of omega-automata: Complexity and connection to second order logic, master's thesis, University of Kiel, Germany (1998)
21. Löding, C., Thomas, W.: Alternating automata and logics over infinite words. In: Watanabe, O., Hagiya, M., Ito, T., van Leeuwen, J., Mosses, P.D. (eds.) TCS 2000. LNCS, vol. 1872. Springer, Heidelberg (2000)
22. Miyano, S., Hayashi, T.: Alternating finite automata on ω-words. Theoret. Comput. Sci. 32 (1984)
23. Muller, D., Saoudi, A., Schupp, P.: Alternating automata, the weak monadic theory of trees and its complexity. Theoret. Comput. Sci. 97 (1992)
24. Muller, D., Schupp, P.: Alternating automata on infinite trees. Theoret. Comput. Sci. 54 (1987)
25. Muller, D., Schupp, P.: Simulating alternating tree automata by nondeterministic automata: New results and new proofs of the theorems of Rabin, McNaughton and Safra. Theoret. Comput. Sci. 141 (1995)

26. Pnueli, A.: The temporal logic of programs. In: Proc. of FOCS 1977 (1977)
27. Rohde, G.: Alternating automata and the temporal logic of ordinals, PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA (1997)
28. Vardi, M.: A temporal fixpoint calculus. In: Proc. of POPL 1988 (1988)
29. Vardi, M.: A note on the reduction of two-way automata to one-way automata. Inform. Process. Lett. 30 (1989)
30. Vardi, M.: Reasoning about the past with two-way automata. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443. Springer, Heidelberg (1998)
31. Vardi, M.: Automata-theoretic model checking revisited. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349. Springer, Heidelberg (2007)

Discounted Properties of Probabilistic Pushdown Automata*

Tomáš Brázdil, Václav Brožek, Jan Holeček, and Antonín Kučera

Faculty of Informatics, Masaryk University,
Botanická 68a, 60200 Brno,
Czech Republic
{brazdil,brozek,holecek,kucera}@fi.muni.cz

Abstract. We show that several basic discounted properties of probabilistic pushdown automata related both to terminating and non-terminating runs can be efficiently approximated up to an arbitrarily small given precision.

1 Introduction

Discounting formally captures the natural intuition that the far-away future is not as important as the near future. In the discrete time setting, the discount assigned to a state visited after k time units is λ^k, where $0 < \lambda < 1$ is a fixed constant. Thus, the "weight" of states visited lately becomes progressively smaller. Discounting (or inflation) is a key paradigm in economics and has been studied in Markov decision processes as well as game theory [18,15]. More recently, discounting has been found appropriate also in systems theory (see, e.g., [6]), where it allows to put more emphasis on events that occur early. For example, even if a system is guaranteed to handle every request eventually, it still makes a big difference whether the request is handled early or lately, and discounting provides a convenient formalism for specifying and even quantifying this difference.

In this paper, we concentrate on basic discounted properties of probabilistic pushdown automata (pPDA), which provide a natural model for probabilistic systems with unbounded recursion [9,4,10,3,14,12,13]. Thus, we aim at filling a gap in our knowledge on probabilistic PDA, which has so far been limited only to non-discounted properties. As the main result, we show that several fundamental discounted properties related to long-run behaviour of probabilistic PDA (such as the discounted gain or the total discounted accumulated reward) are expressible as the least solutions of efficiently constructible systems of recursive monotone polynomial equations whose form admits the application of the recent results [17,8] about a fast convergence of Newton's approximation method. This entails the decidability of the corresponding *quantitative problems* (we ask whether the value of a given discounted long-run property is equal to or bounded by a given rational constant). A more important consequence is that the discounted long-run properties are *computational* in the sense that they can be efficiently

* Supported by the research center Institute for Theoretical Computer Science (ITI), project No. 1M0545.

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 230–242, 2008.

approximated up to an arbitrarily small given precision. This is very different from the non-discounted case, where the respective quantitative problems are also decidable but no efficient approximation schemes are known[1]. This shows that discounting, besides its natural practical appeal, has also mathematical and computational benefits.

We also consider discounted properties related to terminating runs, such as the discounted termination probability and the discounted reward accumulated along a terminating run. Further, we examine the relationship between the discounted and non-discounted variants of a given property. Intuitively, one expects that a discounted property should be close to its non-discounted variant as the discount approaches 1. This intuition is mostly confirmed, but in some cases the actual correspondence is more complicated.

Concerning the level of originality of the presented work, the results about terminating runs are obtained as simple extensions of the corresponding results for the non-discounted case presented in [9,14,10]. New insights and ideas are required to solve the problems about discounted long-run properties of probabilistic PDA (the discounted gain and the total discounted accumulated reward), and also to establish the correspondence between these properties and their non-discounted versions. A more detailed discussion and explanation is postponed to Sections 3 and 4.

Since most of the proofs are rather technical, they are not included in this paper. The main body of the paper contains the summary of the main results, and the relationship to the previous work is carefully discussed at appropriate places. Proofs in full detail can be found in [2].

2 Basic Definitions

In this paper, we use $\mathbb{N}$, $\mathbb{N}_0$, $\mathbb{Q}$, and $\mathbb{R}$ to denote the sets of positive integers, nonnegative integers, rational numbers, and real numbers, respectively. We also use the standard notation for intervals of real numbers, writing, e.g., $(0,1]$ to denote the set $\{x \in \mathbb{R} \mid 0 < x \leq 1\}$.

The set of all finite words over a given alphabet Σ is denoted Σ^*, and the set of all infinite words over Σ is denoted Σ^ω. We also use Σ^+ to denote the set $\Sigma^* \setminus \{\varepsilon\}$ where ε is the empty word. The length of a given $w \in \Sigma^* \cup \Sigma^\omega$ is denoted $len(w)$, where the length of an infinite word is ω. Given a word (finite or infinite) over Σ, the individual letters of w are denoted $w(0), w(1), \cdots$.

Let $V \neq \emptyset$, and let $\rightarrow \subseteq V \times V$ be a *total* relation (i.e., for every $v \in V$ there is some $u \in V$ such that $v \rightarrow u$). The reflexive and transitive closure of $\rightarrow$ is denoted $\rightarrow^*$. A *path* in $(V, \rightarrow)$ is a finite or infinite word $w \in V^+ \cup V^\omega$ such that $w(i-1) \rightarrow w(i)$ for every $1 \leq i < len(w)$. A *run* in $(V, \rightarrow)$ is an infinite path in V. The set of all runs that start with a given finite path w is denoted $Run(w)$.

A *probability distribution* over a finite or countably infinite set X is a function $f : X \rightarrow [0,1]$ such that $\sum_{x \in X} f(x) = 1$. A probability distribution f over X is *positive* if $f(x) > 0$ for every $x \in X$, and *rational* if $f(x) \in \mathbb{Q}$ for every $x \in X$. A *σ-field* over a set Ω is a set $\mathcal{F} \subseteq 2^\Omega$ that includes Ω and is closed under complement and countable union.

[1] For example, the existing approximation methods for the (non-discounted) gain employ the decision procedure for the existential fragment of $(\mathbb{R}, +, *, \leq)$, which is rather inefficient.

A *probability space* is a triple $(\Omega, \mathcal{F}, \mathcal{P})$ where Ω is a set called *sample space*, $\mathcal{F}$ is a σ-field over Ω whose elements are called *events*, and $\mathcal{P} : \mathcal{F} \to [0,1]$ is a *probability measure* such that, for each countable collection $\{X_i\}_{i\in I}$ of pairwise disjoint elements of $\mathcal{F}$ we have that $\mathcal{P}(\bigcup_{i\in I} X_i) = \sum_{i\in I} \mathcal{P}(X_i)$, and moreover $\mathcal{P}(\Omega)$=1. A *random variable* over a probability space $(\Omega, \mathcal{F}, \mathcal{P})$ is a function $X : \Omega \to \mathbb{R} \cup \{\bot\}$, where $\bot \notin \mathbb{R}$ is a special "undefined" symbol, such that $\{\omega \in \Omega \mid X(\omega) \leq c\} \in \mathcal{F}$ for every $c \in \mathbb{R}$. If $\mathcal{P}(X{=}\bot) = 0$, then the *expected value* of X is defined by $\mathbb{E}[X] = \int_{\omega\in\Omega} X(\omega)\, \mathrm{d}\mathcal{P}$.

Definition 1 (Markov Chain). *A* Markov chain *is a triple $M = (S, \to, Prob)$ where S is a finite or countably infinite set of* states*, $\to \subseteq S \times S$ is a total* transition relation*, and Prob is a function which to each state $s \in S$ assigns a positive probability distribution over the outgoing transitions of s. As usual, we write $s \xrightarrow{x} t$ when $s \to t$ and x is the probability of $s \to t$.*

To every $s \in S$ we associate the probability space $(Run(s), \mathcal{F}, \mathcal{P})$ where $\mathcal{F}$ is the σ-field generated by all *basic cylinders* $Run(w)$ where w is a finite path starting with s, and $\mathcal{P} : \mathcal{F} \to [0,1]$ is the unique probability measure such that $\mathcal{P}(Run(w)) = \Pi_{i=1}^{len(w)-1} x_i$ where $w(i{-}1) \xrightarrow{x_i} w(i)$ for every $1 \leq i < len(w)$. If $len(w) = 1$, we put $\mathcal{P}(Run(w)) = 1$.

Definition 2 (probabilistic PDA). *A* probabilistic pushdown automaton (pPDA) *is a tuple $\Delta = (Q, \Gamma, \delta, Prob)$ where Q is a finite set of* control states*, Γ is a finite* stack alphabet*, $\delta \subseteq (Q \times \Gamma) \times (Q \times \Gamma^{\leq 2})$ is a* transition relation *(here $\Gamma^{\leq 2} = \{w \in \Gamma^* \mid len(w) \leq 2\}$), and $Prob : \delta \to (0,1]$ is a rational* probability assignment *such that for all $pX \in Q \times \Gamma$ we have that $\sum_{pX\to q\alpha} Prob(pX \to q\alpha) = 1$.*

A configuration *of Δ is an element of $Q \times \Gamma^*$, and the set of all configurations of Δ is denoted $C(\Delta)$.*

To each pPDA $\Delta = (Q, \Gamma, \delta, Prob_\Delta)$ we associate a Markov chain $M_\Delta = (C(\Delta), \to, Prob)$, where $p\varepsilon \xrightarrow{1} p\varepsilon$ for every $p \in Q$, and $pX\beta \xrightarrow{x} q\alpha\beta$ iff $(pX, q\alpha) \in \delta$, $Prob_\Delta(pX \to q\alpha) = x$, and $\beta \in \Gamma^*$. For all $p, q \in Q$ and $X \in \Gamma$, we use $Run(pXq)$ to denote the set of all $w \in Run(pX)$ such that $w(n) = q\varepsilon$ for some $n \in \mathbb{N}$, and $Run(pX{\uparrow})$ to denote the set $Run(pX) \setminus \bigcup_{q\in Q} Run(pXq)$. The runs of $Run(pXq)$ and $Run(pX{\uparrow})$ are sometimes referred to as *terminating* and *non-terminating*, respectively.

3 Discounted Properties of Probabilistic PDA

In this section we introduce the family of discounted properties of probabilistic PDA studied in this paper. These notions are not PDA-specific and could be defined more abstractly for arbitrary Markov chains. Nevertheless, the scope of our study is limited to probabilistic PDA, and the notation becomes more suggestive when it directly reflects the structure of a given pPDA.

For the rest of this section, we fix a pPDA $\Delta = (Q, \Gamma, \delta, Prob_\Delta)$, a nonnegative *reward function* $f : Q \times \Gamma \to \mathbb{Q}$, and a *discount function* $\lambda : Q \times \Gamma \to [0,1)$. The functions f and λ are extended to all elements of $Q \times \Gamma^+$ by stipulating that $f(pX\alpha) = f(pX)$ and $\lambda(pX\alpha) = \lambda(pX)$, respectively. One can easily generalize the presented arguments also to rewards and discounts that depend on the whole stack content, provided that this

dependence is "regular", i.e., can be encoded by a finite-state automaton which reads the stack bottom-up. This extension is obtained just by applying standard techniques that have been used in, e.g., [11]. Also note that λ can assign a different discount to each element of $Q \times \Gamma$, and that the discount can also be 0. Hence, we in fact work with a slightly generalized form of discounting which can also reflect relative speed of transitions.

We start by defining several simple random variables. The definitions are parametrized by the functions f and λ, control states $p, q \in Q$, and a stack symbol $X \in \Gamma$. For every run w and $i \in \mathbb{N}_0$, we use $\lambda(w^i)$ to denote $\Pi_{j=0}^{i} \lambda(w(j))$, i.e., the discount accumulated up to $w(i)$. Note that the initial state of w is also discounted, which is somewhat non-standard but technically convenient (the equations constructed in Section 4 become more readable).

$$I_{pXq}(w) = \begin{cases} 1 & \text{if } w \in \mathit{Run}(pXq) \\ 0 & \text{otherwise} \end{cases}$$

$$I^{\lambda}_{pXq}(w) = \begin{cases} \lambda(w^{n-1}) & \text{if } w \in \mathit{Run}(pXq),\ w(n-1) \neq w(n) = q\varepsilon \\ 0 & \text{otherwise} \end{cases}$$

$$R^{f}_{pXq}(w) = \begin{cases} \sum_{i=0}^{n-1} f(w(i)) & \text{if } w \in \mathit{Run}(pXq),\ w(n-1) \neq w(n) = q\varepsilon \\ 0 & \text{otherwise} \end{cases}$$

$$R^{f,\lambda}_{pXq}(w) = \begin{cases} \sum_{i=0}^{n-1} \lambda(w^i) \cdot f(w(i)) & \text{if } w \in \mathit{Run}(pXq),\ w(n-1) \neq w(n) = q\varepsilon \\ 0 & \text{otherwise} \end{cases}$$

$$G^{f}_{pX}(w) = \begin{cases} \lim_{n\to\infty} \frac{\sum_{i=0}^{n} f(w(i))}{n+1} & \text{if } w \in \mathit{Run}(pX\uparrow) \text{ and the limit exists} \\ \bot & \text{otherwise} \end{cases}$$

$$G^{f,\lambda}_{pX}(w) = \begin{cases} \lim_{n\to\infty} \frac{\sum_{i=0}^{n} \lambda(w^i) \cdot f(w(i))}{\sum_{i=0}^{n} \lambda(w^i)} & \text{if } w \in \mathit{Run}(pX\uparrow) \text{ and the limit exists} \\ \bot & \text{otherwise} \end{cases}$$

$$X^{f,\lambda}_{pX}(w) = \begin{cases} \sum_{i=0}^{\infty} \lambda(w^i) f(w(i)) & \text{if } w \in \mathit{Run}(pX\uparrow) \\ 0 & \text{otherwise} \end{cases}$$

The variable I_{pXq} is a simple indicator telling whether a given run belongs to $\mathit{Run}(pXq)$ or not. Hence, $\mathbb{E}[I_{pXq}]$ is the probability of $\mathit{Run}(pXq)$, i.e., the probability of all runs $w \in \mathit{Run}(pX)$ that terminate in $q\varepsilon$. The variable I^{λ}_{pXq} is the discounted version of I_{pXq}, and its expected value can thus be interpreted as the "discounted termination probability", where more weight is put on terminated states visited early. Hence, $\mathbb{E}[I^{\lambda}_{pXq}]$ is a meaningful value which can be used to quantify the difference between two configurations with the same termination probability but different termination time. From now on, we write $[pXq]$ and $[pXq, \lambda]$ instead of $\mathbb{E}[I_{pXq}]$ and $\mathbb{E}[I^{\lambda}_{pXq}]$, respectively, and we also use $[pX\uparrow]$ to denote $1 - \sum_{q\in Q}[pXq]$. The computational aspects of $[pXq]$ have been examined in [9,14], where it is shown that the family of all $[pXq]$ forms the least solution of an effectively constructible system of monotone polynomial equations. By

applying the recent results [17,8] about a fast convergence of Newton's method, it is possible to approximate the values of $[pXq]$ efficiently (the precise values of $[pXq]$ can be irrational). In Section 4, we generalize these results to $[pXq, \lambda]$.

The variable R^f_{pXq} returns to every $w \in Run(pXq)$ the total f-reward accumulated up to $q\varepsilon$. For example, if $f(rY) = 1$ for every $rY \in Q \times \Gamma$, then the variable returns the number of transitions executed before hitting the configuration $q\varepsilon$. In [10], the *conditional* expected value $\mathbb{E}[R^f_{pXq} \mid Run(pXq)]$ has been studied in detail. This value can be used to analyze important properties of terminating runs; for example, if f is as above, then

$$\sum_{q \in Q} [pXq] \cdot \mathbb{E}[R^f_{pXq} \mid Run(pXq)]$$

is the conditional expected termination time of a run initiated in pX, under the condition that the run terminates (i.e., the stack is eventually emptied). In [10], it has been shown that the family of all $\mathbb{E}[R^f_{pXq} \mid Run(pXq)]$ forms the least solution of an effectively constructible system of recursive polynomial equations. One disadvantage of $\mathbb{E}[R^f_{pXq} \mid Run(pXq)]$ (when compared to $[pXq, \lambda]$ which also reflects the length of terminating runs) is that this conditional expected value can be infinite even in situations when $[pXq] = 1$.

The discounted version $R^{f,\lambda}_{pXq}$ of R^f_{pXq} assigns to each $w \in Run(pXq)$ the total discounted reward accumulated up to $q\varepsilon$. In Section 4, we extend the aforementioned results about $\mathbb{E}[R^f_{pXq} \mid Run(pXq)]$ to the family of all $\mathbb{E}[R^{f,\lambda}_{pXq} \mid Run(pXq)]$. The extension is actually based on analyzing the properties of the (unconditional) expected value $\mathbb{E}[R^{f,\lambda}_{pXq}]$. At first glance, $\mathbb{E}[R^{f,\lambda}_{pXq}]$ does not seem to provide any useful information, at least in situations when $[pXq] \neq 1$. However, this expected value can be used to express not only $\mathbb{E}[R^{f,\lambda}_{pXq} \mid Run(pXq)]$, but also other properties such as $\mathbb{E}[G^{f,\lambda}_{pX}]$ or $\mathbb{E}[X^{f,\lambda}_{pX}]$ discussed below, and can be effectively approximated by Newton's method. Hence, the variable $R^{f,\lambda}_{pXq}$ and its expected value provide a crucial technical tool for solving the problems of our interest.

The variable G^f_{pX} assigns to each non-terminating run its average reward per transition, provided that the corresponding limit exists. For finite-state Markov chains, the average reward per transition exists for almost all runs, and hence the corresponding expected value (also called *the gain*[2]) always exists. In the case of pPDA, it can happen that $\mathcal{P}(G^f_{pX}{=}\perp) > 0$ even if $[pX\uparrow] = 1$, and hence the gain $\mathbb{E}[G^f_{pX}]$ does not necessarily exist. A concrete example is given in Section 4. In [10], it has been shown that if all $\mathbb{E}[R^g_{tYs} \mid Run(tYs)]$ are finite (where $g(rZ) = 1$ for all $rZ \in Q \times \Gamma$), then the gain is guaranteed to exist and can be effectively expressed in first order theory of the reals. This result relies on a construction of an auxiliary finite-state Markov chain with possibly irrational transition probabilities, and this method does not allow for efficient approximation of the gain.

In Section 4, we examine the properties of the *discounted gain* $\mathbb{E}[G^{f,\lambda}_{pX}]$ which are remarkably different from the aforementioned properties of the gain (these results constitute the first highlight of our paper). First, we *always* have that $\mathcal{P}(G^{f,\lambda}_{pX}{=}\perp \mid Run(pX\uparrow)) = 0$ whenever $[pX\uparrow] > 0$, and hence the discounted gain is guaranteed to exist whenever $[pX\uparrow] = 1$. Further, we show that the discounted gain

[2] The gain is one of the fundamental concepts in performance analysis.

can be efficiently approximated by Newton's method. One intuitively expects that the discounted gain is close to the value of the gain as the discount approaches 1, and we show that this is indeed the case when the gain exists (the proof is not trivial). Thus, we obtain alternative proofs for some of the results about the gain that have been presented in [10], but unfortunately we do not yield an efficient approximation scheme for the (non-discounted) gain, because we were not able to analyze the corresponding convergence rate. More details are given in Section 4.

The variable $X_{pX}^{f,\lambda}$ assigns to each non-terminating run the total discounted reward accumulated along the whole run. Note that the corresponding infinite sum always exists and it is finite. If $[pX\uparrow] = 1$, then the expected value $\mathbb{E}[X_{pX}^{f,\lambda}]$ exactly corresponds to the expected discounted payoff, which is a fundamental and deeply studied concept in stochastic programming (see, e.g., [18,15]). In Section 4, we show that the family of all $\mathbb{E}[X_{pX}^{f,\lambda}]$ forms the least solution of an effectively constructible system of monotone polynomial equations. Hence, these expected values can also be effectively approximated by Newton's method by applying the results of [17,8]. We also show how to express $\mathbb{E}[X_{pX}^{f,\lambda} \mid Run(pX\uparrow)]$, which is more relevant in situations when $0 < [pX\uparrow] < 1$.

4 Computing the Discounted Properties of Probabilistic PDA

In this section we show that the (conditional) expected values of the discounted random variables introduced in Section 3 are expressible as the least solutions of efficiently constructible systems of recursive equations. This allows to encode these values in first order theory of the reals, and design efficient approximation schemes for some of them.

Recall that first order theory of the reals $(\mathbb{R}, +, *, \leq)$ is decidable [19], and the existential fragment is even solvable in polynomial space [5]. The following definition explains what we mean by encoding a certain value in $(\mathbb{R}, +, *, \leq)$.

Definition 3. *We say that some $c \in \mathbb{R}$ is* encoded *by a formula $\Phi(x)$ of $(\mathbb{R}, +, *, \leq)$ iff the formula $\forall x.(\Phi(x) \Leftrightarrow x{=}c)$ holds.*

Note that if a given $c \in \mathbb{R}$ is encoded by $\Phi(x)$, then the problems whether $c = \varrho$ and $c \leq \varrho$ for a given rational constant ϱ are decidable (we simply check the validity of the formulae $\Phi(x/\varrho)$ and $\exists x.(\Phi(x) \wedge x \leq \varrho)$, respectively).

For the rest of this section, we fix a pPDA $\Delta = (Q, \Gamma, \delta, Prob_\Delta)$, a nonnegative *reward function* $f : Q \times \Gamma \to \mathbb{Q}$, and a *discount function* $\lambda : Q \times \Gamma \to [0, 1)$. As a warm-up, let us first consider the family of expected values $[pXq, \lambda]$. For each of them, we fix the corresponding first order variable $\langle\!\langle \boldsymbol{pXq, \lambda} \rangle\!\rangle$, and construct the following equation (for the sake of readability, each variable occurrence is typeset in boldface):

$$\langle\!\langle \boldsymbol{pXq, \lambda} \rangle\!\rangle = \sum_{pX \xrightarrow{x} q\varepsilon} x \cdot \lambda(pX) + \sum_{pX \xrightarrow{x} rY} x \cdot \lambda(pX) \cdot \langle\!\langle \boldsymbol{rYq, \lambda} \rangle\!\rangle + \sum_{pX \xrightarrow{x} rYZ,\ s \in Q} x \cdot \lambda(pX) \cdot \langle\!\langle \boldsymbol{rYs, \lambda} \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{sZq, \lambda} \rangle\!\rangle \tag{1}$$

Thus, we produce a finite system of recursive equations (S1). This system is rather similar to the system for termination probabilities $[pXq]$ constructed in [9,14]. The only modification is the introduction of the discount factor $\lambda(pX)$. A proof of the following theorem is also just a technical extension of the proof given in [9,14] (see [2]).

Theorem 4. *The tuple of all $[pXq, \lambda]$ is the least nonnegative solution of the system (S1).*

Now consider the expected value $\mathbb{E}[R^{f,\lambda}_{pXq}]$. For all $p, q \in Q$ and $X \in \Gamma$ we fix a first order variable $\langle\!\langle \boldsymbol{pXq} \rangle\!\rangle$ and construct the following equation:

$$\langle\!\langle \boldsymbol{pXq} \rangle\!\rangle = \sum_{pX \xrightarrow{x} q\varepsilon} x \cdot \lambda(pX) \cdot f(pX) + \sum_{pX \xrightarrow{x} rY} x \cdot \lambda(pX) \cdot ([rYq] \cdot f(pX) + \langle\!\langle \boldsymbol{rYq} \rangle\!\rangle)$$
$$+ \sum_{pX \xrightarrow{x} rYZ,\ s \in Q} x \cdot \lambda(pX) \cdot ([rYs] \cdot [sZq] \cdot f(pX) + [sZq] \cdot \langle\!\langle \boldsymbol{rYs} \rangle\!\rangle + [rYs, \lambda] \cdot \langle\!\langle \boldsymbol{sZq} \rangle\!\rangle) \tag{2}$$

Thus, we obtain the system (S2). Note that termination probabilities and discounted termination probabilities are treated as "known constants" in the equations of (S2).

As opposed to (S1), the equations of system (S2) do not have a good intuitive meaning. At first glance, it is not clear why these equations should hold, and a formal proof of this fact requires advanced arguments. The proof of the following theorem is already non-trivial (see [2]).

Theorem 5. *The tuple of all $\mathbb{E}[R^{f,\lambda}_{pXq}]$ is the least nonnegative solution of the system (S2).*

The conditional expected values $\mathbb{E}[R^{f,\lambda}_{pXq} \mid Run(pXq)]$ make sense only if $[pXq] > 0$, which can be checked in time polynomial in the size of Δ because $[pXq] > 0$ iff $pX \rightarrow^* q\varepsilon$, and the reachability problem for PDA is in **P** (see, e.g., [7]). The next theorem says how to express $\mathbb{E}[R^{f,\lambda}_{pXq} \mid Run(pXq)]$ using $\mathbb{E}[R^{f,\lambda}_{pXq}]$.

Theorem 6. *For all $p, q \in Q$ and $X \in \Gamma$ such that $[pXq] > 0$ we have that*

$$\mathbb{E}[R^{f,\lambda}_{pXq} \mid Run(pXq)] = \frac{\mathbb{E}[R^{f,\lambda}_{pXq}]}{[pXq]} \tag{3}$$

Now we turn our attention to the discounted long-run properties of probabilistic PDA introduced in Section 3. These results represent the core of our paper.

As we already mentioned, the system (S1) can also be used to express the family of termination probabilities $[pXq]$. This is achieved simply by replacing each $\lambda(pX)$ with 1 (thus, we obtain the equational systems originally presented in [9,14]). Hence, we can also express the probability of *non-termination*:

$$[pX\uparrow] = 1 - \sum_{q \in Q} [pXq] \tag{4}$$

Note that this equation *is not monotone* (by increasing $[pXq]$ we decrease $[pX\uparrow]$), which leads to some complications discussed in Section 4.1.

Now we have all the tools that are needed to construct an equational system for the family of all $\mathbb{E}[X^{f,\lambda}_{pX}]$. For all $p \in Q$ and $X \in \Gamma$, we fix a first order variable $\langle\!\langle pX \rangle\!\rangle$ and construct the following equation, which gives us the system (S5):

$$\begin{aligned}\langle\!\langle pX \rangle\!\rangle = &\sum_{pX \xrightarrow{x} rY} x \cdot \lambda(pX) \cdot ([rY\uparrow] \cdot f(pX) + \langle\!\langle rY \rangle\!\rangle) \\ &+ \sum_{pX \xrightarrow{x} rYZ} x \cdot \lambda(pX) \cdot ([rY\uparrow] \cdot f(pX) + \langle\!\langle rY \rangle\!\rangle) \\ &+ \sum_{pX \xrightarrow{x} rYZ,\ s \in Q} x \cdot \lambda(pX) \cdot \Big([rYs] \cdot [sZ\uparrow] \cdot f(pX) + [sZ\uparrow] \cdot \mathbb{E}[R^{f,\lambda}_{rYs}] + [rYs, \lambda] \cdot \langle\!\langle sZ \rangle\!\rangle\Big)\end{aligned} \tag{5}$$

The equations of (S5) are even less readable than the ones of (S2). However, note that the equations are monotone and efficiently constructible. The proof of the following theorem is based on advanced arguments.

Theorem 7. *The tuple of all $\mathbb{E}[X^{f,\lambda}_{pX}]$ is the least nonnegative solution of the system (S5).*

In situations when $[pX\uparrow] < 1$, $\mathbb{E}[X^{f,\lambda}_{pX} \mid Run(pX\uparrow)]$ may be more relevant than $\mathbb{E}[X^{f,\lambda}_{pX}]$. The next theorem says how to express this expected value.

Theorem 8. *For all $p, q \in Q$ and $X \in \Gamma$ such that $[pX\uparrow] > 0$ we have that*

$$\mathbb{E}[X^{f,\lambda}_{pX} \mid Run(pX\uparrow)] = \frac{\mathbb{E}[X^{f,\lambda}_{pX}]}{[pX\uparrow]} \tag{6}$$

Concerning the equations of (S6), there is one notable difference from all of the previous equational systems. The only known method of solving the problem whether $[pX\uparrow] > 0$ employs the decision procedure for the existential fragment of $(\mathbb{R}, +, *, \leq)$, and hence the best known upper bound for this problem is **PSPACE**. This means that the equations of (S6) cannot be constructed efficiently, because there is no efficient way of determining all p, q and X such that $[pX\uparrow] > 0$.

The last discounted property of probabilistic PDA which is to be investigated is the discounted gain $\mathbb{E}[G^{f,\lambda}_{pX}]$. Here, we only managed to solve the special case when λ is a constant function.

Theorem 9. *Let λ be a constant discount function such that $\lambda(rY) = \kappa$ for all $rY \in Q \times \Gamma$, and let $p \in Q$, $X \in \Gamma$ such that $[pX\uparrow] = 1$. Then*

$$\mathbb{E}[G^{f,\lambda}_{pX}] = (1 - \kappa) \cdot \mathbb{E}[X^{f,\lambda}_{pX}] \tag{7}$$

A proof of Theorem 9 is simple. Let $w \in Run(pX\uparrow)$. Since both $\lim_{n\to\infty} \sum_{i=0}^{n} \lambda(w^i)$ and $\lim_{n\to\infty} \sum_{i=0}^{n} \lambda(w^i) f(w(i))$ exist and the latter is equal to $(1-\kappa)^{-1}$ the claim follows from the linearity of the expected value.

Note that the equations of (S7) can be constructed efficiently (in polynomial time), because the question whether $[pX\uparrow] = 1$ is equivalent to checking whether $[pXq] = 0$

for all $q \in Q$, which is equivalent to checking whether $pX \not\rightarrow^* q\varepsilon$ for all $q \in Q$. Hence, it suffices to apply a polynomial-time decision procedure for PDA reachability such as [7].

Since all equational systems constructed in this section contain just summation, multiplication, and division, one can easily encode all of the considered discounted properties in $(\mathbb{R}, +, *, \leq)$ in the sense of Definition 3. For a given discounted property c, the corresponding formula $\Phi(x)$ looks as follows:

$$\exists \boldsymbol{v} \left(\mathit{solution}(\boldsymbol{v}) \ \wedge \ (\forall \boldsymbol{u} \ (\mathit{solution}(\boldsymbol{u}) \Rightarrow \boldsymbol{v} \leq \boldsymbol{u}) \ \wedge \ x = \boldsymbol{v}_i \right)$$

Here $\boldsymbol{v}$ and $\boldsymbol{u}$ are tuples of fresh first order variables that correspond (in one-to-one fashion) to the variables employed in the equational systems (S1), (S2), (S3), (S4), (S5), (S6), and (S7). The subformulae $\mathit{solution}(\boldsymbol{v})$ and $\mathit{solution}(\boldsymbol{u})$ say that the variables of $\boldsymbol{v}$ and $\boldsymbol{u}$ form a solution of the equational systems (S1), (S2), (S3), (S4), (S5), (S6), and (S7). Note that the subformulae $\mathit{solution}(\boldsymbol{v})$ and $\mathit{solution}(\boldsymbol{u})$ are indeed expressible in $(\mathbb{R}, +, *, \leq)$, because the right-hand sides of all equational systems contain just summation, multiplication, division, and employ only constants that themselves correspond to some variables in $\boldsymbol{v}$ or $\boldsymbol{u}$. The $\boldsymbol{v}_i$ is the variable of $\boldsymbol{v}$ which corresponds to the considered property c, and the x is the only free variable of the formula $\Phi(x)$. Note that $\Phi(x)$ can be constructed in space which is polynomial in the size of a given pPDA Δ (the main cost is the construction of the system (S6)), but the length of $\Phi(x)$ is only polynomial in the size of Δ, λ, and f. Since the alternation depth of quantifiers in $\Phi(x)$ is fixed, we can apply the result of [16] which says that every fragment of $(\mathbb{R}, +, *, \leq)$ where the alternation depth of quantifiers is bounded by a fixed constant is decidable in exponential time. Thus, we obtain the following theorem:

Theorem 10. *Let c be one of the discounted properties of pPDA considered in this section, i.e., c is either $[pXq, \lambda]$, $\mathbb{E}[R^{f,\lambda}_{pXq}]$, $\mathbb{E}[R^{f,\lambda}_{pXq} \mid \mathit{Run}(pXq)]$, $\mathbb{E}[X^{f,\lambda}_{pX}]$, $\mathbb{E}[X^{f,\lambda}_{pX} \mid \mathit{Run}(pX\uparrow)]$, or $\mathbb{E}[G^{f,\lambda}_{pX}]$ (in the last case we further require that λ is constant). The problems whether $c = \varrho$ and $c \leq \varrho$ for a given rational constant ϱ are in **EXPTIME**.*

The previous theorem extends the results achieved in [9,10,14] to discounted properties of pPDA. However, in the case of discounted long-run properties $\mathbb{E}[X^{f,\lambda}_{pX}]$, $\mathbb{E}[X^{f,\lambda}_{pX} \mid \mathit{Run}(pX\uparrow)]$, and $\mathbb{E}[G^{f,\lambda}_{pX}]$, the presented proof is completely different from the non-discounted case. Moreover, the constructed equations take the form which allows to design efficient approximation scheme for these values, and this is what we show in the next subsection.

4.1 The Application of Newton's Method

In this section we show how to apply the recent results [17,8] about fast convergence of Newton's method for systems of monotone polynomial equations to the discounted properties introduced in Section 3. We start by recalling some notions and results presented in [17,8].

Monotone systems of polynomial equations (MSPEs) are systems of fixed point equations of the form $x_1 = f_1(x_1, \cdots, x_n), \ \cdots \ , x_n = f_n(x_1, \cdots, x_n)$, where each f_i

is a polynomial with nonnegative real coefficients. Written in vector form, the system is given as $\boldsymbol{x} = f(\boldsymbol{x})$, and solutions of the system are exactly the fixed points of f. To f we associate the directed graph H_f where the nodes are the variables $x_1, \ldots, x_n$ and (x_i, x_j) is an edge iff x_j appears in f_i. A subset of equations is a *strongly connected component (SCC)* if its associated subgraph is a SCC of H_f.

Observe that each of the systems (S1), (S2), and (S5) forms a MSPE. Also observe that the system (S1) uses only simple coefficients obtained by multiplying transition probabilities of Δ with the return values of λ, while the coefficients in (S2) and (S5) are more complicated and also employ constants such as $[rYq]$, $[rYs, \lambda]$, $\mathbb{E}[R^{f,\lambda}_{pXq}]$, or $[rY\uparrow]$.

The problem of finding the least nonnegative solution of a given MSPE $\boldsymbol{x} = f(\boldsymbol{x})$ can be obviously reduced to the problem of finding the least nonnegative solution for $F(\boldsymbol{x}) = \boldsymbol{0}$, where $F(\boldsymbol{x}) = f(\boldsymbol{x}) - \boldsymbol{x}$. The Newton's method for approximating the least solution of $F(\boldsymbol{x}) = \boldsymbol{0}$ is based on computing a sequence $\boldsymbol{x}^{(0)}, \boldsymbol{x}^{(1)}, \cdots$, where $\boldsymbol{x}^{(0)} = \boldsymbol{0}$ and

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^k - (F'(\boldsymbol{x}^k))^{-1} F(\boldsymbol{x}^k)$$

where $F'(\boldsymbol{x})$ is the Jacobian matrix of partial derivatives. If the graph H_f is strongly connected, then the method is guaranteed to converge *linearly* [17,8]. This means that there is a threshold k_f such that after the initial k_f iterations of the Newton's method, each additional bit of precision requires only 1 iteration. In [8], an upper bound for k_f is presented.

For general MSPE where H_f is not necessarily strongly connected, a more structured method called *Decomposed Newton's method (DNM)* can be used. Here, the component graph of H_f is computed and the SCCs are divided according to their depth. DNM proceeds bottom-up by computing $k \cdot 2^t$ iterations of Newton's method for each of the SCCs of depth t, where t goes from the height of the component graph to 0. After computing the approximations for the SCCs of depth i, the computed values are fixed, the corresponding equations are removed, and the SCCs of depth $i-1$ are processed in the same way, using the previously fixed values as constants. This goes on until all SCCs are processed. It was demonstrated in [14] that DNM is guaranteed to converge to the least solution as k increases. In [17], it was shown that DNM is even guaranteed to converge linearly. Note, however, that the number of iterations of the original Newton's method in one iteration of DNM is exponential in the depth of the component graph of H_f.

Now we show how to apply these results to the systems (S1), (S2), and (S5). First, we also add a system (S0) whose least solution is the tuple of all termination probabilities $[pXq]$ (the system (S0) is very similar to the system (S1), the only difference is that each $\lambda(pX)$ is replaced with 1). The systems (S0), (S1), (S2), and (S5) themselves are not necessarily strongly connected, and we use H to denote the height of the component graph of (S0). Note that the height of the component graph of (S1), (S2), and (S5) is at most H. Now, we unify the systems (S0), (S1), (S2), and (S5) into one equation system S. What we obtain is a MSPE with three types of coefficients: transition probabilities of Δ, the return values of λ, and non-termination probabilities of the form $[rY\uparrow]$ (the system (S4) cannot be added to S because the resulting system would not be monotone). Observe that

- (S0) and (S1) only use the transition probabilities of Δ and the return values of λ as coefficients;

- (S2) also uses the values defined by (S0) and (S1) as coefficients;
- (S5) uses the values defined by (S0), (S1) and (S2) as coefficients, and it also uses coefficients of the form $[rY\uparrow]$.

This means that the height of the component graph of S is at most $3H$. Now we can apply the DNM in the way described above, with the following technical modification: after computing the termination probabilities $[rYq]$ (in the system (S0)), we compute an *upper approximation* for each $[rY\uparrow]$ according to equation (4), and then subtract an upper bound for the overall error of this upper approximation bound with the same overall error (here we use the technical results of [8]). In this way, we produce a *lower approximation* for each $[rY\uparrow]$ which is used as a constant when processing the other SCCs. Now we can apply the aforementioned results about DNM.

Note that once the values of $[pXq]$, $[pXq, \lambda]$, $\mathbb{E}[R^{f,\lambda}_{pXq}]$, and $\mathbb{E}[X^{f,\lambda}_{pX}]$ are computed with a sufficient precision, we can also compute the values of $\mathbb{E}[R^{f,\lambda}_{pXq} \mid Run(pXq)]$ and $\mathbb{E}[G^{f,\lambda}_{pX}]$ by equations given in Theorem 6 and Theorem 9, respectively. Thus, we obtain the following:

Theorem 11. *The values of* $[pXq]$, $[pXq, \lambda]$, $\mathbb{E}[R^{f,\lambda}_{pXq}]$, $\mathbb{E}[X^{f,\lambda}_{pX}]$, $\mathbb{E}[R^{f,\lambda}_{pXq} \mid Run(pXq)]$, *and* $\mathbb{E}[G^{f,\lambda}_{pX}]$ *can be approximated using DNM, which is guaranteed to converge linearly. The number of iterations of the Newton's method which is needed to compute one iteration of DNM is exponential in* H.

In practice, the parameter H stays usually small. A typical application area of PDA are recursive programs, where stack symbols correspond to the individual procedures, procedure calls are modeled by pushing new symbols onto the stack, and terminating a procedure corresponds to popping a symbol from the stack. Typically, there are "groups" of procedures that call each other, and these groups then correspond to strongly connected components in the component graph. Long chains of procedures $P_1, \cdots, P_n$, where each P_i can only call P_j for $j > i$, are relatively rare, and this is the only situation when the parameter H becomes large.

4.2 The Relationship between Discounted and Non-discounted Properties

In this section we examine the relationship between the discounted properties introduced in Section 3 and their non-discounted variants. Intuitively, one expects that a discounted property should be close to its non-discounted variant as the discount approaches 1. To formulate this precisely, for every $\kappa \in (0, 1)$ we use λ_κ to denote the constant discount function that always returns κ.

The following theorem is immediate. It suffices to observe that the equational systems for the non-discounted properties are obtained from the corresponding equational systems for discounted properties by substituting all $\lambda(pX)$ with 1.

Theorem 12. *We have that*

- $[pXq] = \lim_{\kappa\uparrow 1}[pXq, \lambda_\kappa]$
- $\mathbb{E}[R^{f}_{pXq}] = \lim_{\kappa\uparrow 1} \mathbb{E}[R^{f,\lambda_\kappa}_{pXq}]$
- $\mathbb{E}[R^{f}_{pXq} \mid Run(pXq)] = \lim_{\kappa\uparrow 1} \mathbb{E}[R^{f,\lambda_\kappa}_{pXq} \mid Run(pXq)]$

The situation with discounted gain $\mathbb{E}[G^{f,\lambda}_{pX}]$ is more complicated. First, let us realize that $\mathbb{E}[G^{f}_{pX}]$ does not necessarily exist even if $[pX\uparrow] = 1$. To see this, consider the pPDA with the following rules:

$$pX \xrightarrow{\frac{1}{2}} pYX, \quad pY \xrightarrow{\frac{1}{2}} pYY, \quad pZ \xrightarrow{\frac{1}{2}} pZZ, \quad pX \xrightarrow{\frac{1}{2}} pZX, \quad pY \xrightarrow{\frac{1}{2}} p\varepsilon, \quad pZ \xrightarrow{\frac{1}{2}} p\varepsilon$$

The reward function f is defined by $f(pX) = f(pY) = 0$ and $f(pZ) = 1$. Intuitively, pX models a one-dimensional symmetric random walk with distinguished zero (X), positive numbers (Z) and negative numbers (Y). Observe that $[pX\uparrow] = 1$. However, one can show that $\mathcal{P}(G^{f}_{pX}{=}\perp) = 1$, which means that $\mathbb{E}[G^{f}_{pX}]$ does not exist. The example is elaborated to a greater detail in [2].

The following theorem says that if the gain *does* exist, then it is equal to the limit of discounted gains as κ approaches 1. The opposite direction, i.e., the question whether the existence of the limit of discounted gains implies the existence of the (non-discounted) gain is left open. The proof of the following theorem is not trivial and relies on several subtle observations (see [2]).

Theorem 13. *If* $\mathbb{E}[G^{f}_{pX}]$ *exists, then* $\mathbb{E}[G^{f}_{pX}] = \lim_{\kappa\uparrow 1} \mathbb{E}[G^{f,\lambda_\kappa}_{pX}]$.

Since $\lim_{\lambda\uparrow 1} \mathbb{E}[G^{f,\lambda}_{pX}]$ can be effectively encoded in first order theory of the reals, we obtain an alternative proof of the result established in [10] saying that the gain is effectively expressible in $(\mathbb{R}, +, *, \leq)$. Actually, we obtain a somewhat *stronger* result, because the formula constructed for $\lim_{\lambda\uparrow 1} \mathbb{E}[G^{f,\lambda}_{pX}]$ encodes the gain whenever it exists, while the (very different) formula constructed in [10] encodes the gain only in situation when a certain sufficient condition (mentioned in Section 3) holds. Unfortunately, Theorem 13 does not yet help us to approximate the gain, because the proof does not give any clue how large κ must be chosen in order to approximate the limit upto a given precision.

5 Conclusions

We have shown that a family of discounted properties of probabilistic PDA can be efficiently approximated by decomposed Newton's method. In some cases, it turned out that the discounted properties are "more computational" than their non-discounted counterparts. An interesting open question is whether the scope of our study can be extended to other discounted properties defined, e.g., in the spirit of [3]. More open questions concern decidability of discounted properties of more complex models like Markov decision process and recursive stochastic games in general.

References

1. Diekert, V., Durand, B. (eds.): STACS 2005. LNCS, vol. 3404. Springer, Heidelberg (2005)
2. Brázdil, T., Brožek, V., Holeček, J., Kučera, A.: Discounted properties of probabilistic pushdown automata. Technical report FIMU-RS-2008-06, Faculty of Informatics, Masaryk University (2008)

3. Brázdil, T., Esparza, J., Kučera, A.: Analysis and prediction of the long-run behavior of probabilistic sequential programs with recursion. In: Proceedings of FOCS 2005, pp. 521–530. IEEE Computer Society Press, Los Alamitos (2005)
4. Brázdil, T., Kučera, A., Stražovský, O.: On the decidability of temporal properties of probabilistic pushdown automata. In: Diekert, V., Durand, B. (eds.) STACS 2005 [1]. LNCS, vol. 3404, pp. 145–157. Springer, Heidelberg (2005)
5. Canny, J.: Some algebraic and geometric computations in PSPACE. In: Proceedings of STOC 1988, pp. 460–467. ACM Press, New York (1988)
6. de Alfaro, L., Henzinger, T., Majumdar, R.: Discounting the future in systems theory. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 1022–1037. Springer, Heidelberg (2003)
7. Esparza, J., Hansel, D., Rossmanith, P., Schwoon, S.: Efficient algorithms for model checking pushdown systems. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 232–247. Springer, Heidelberg (2000)
8. Esparza, J., Kiefer, S., Luttenberger, M.: Convergence thresholds of Newton's method for monotone polynomial equations. In: Proceedings of STACS 2008 (2008)
9. Esparza, J., Kučera, A., Mayr, R.: Model-checking probabilistic pushdown automata. In: Proceedings of LICS 2004, pp. 12–21. IEEE Computer Society Press, Los Alamitos (2004)
10. Esparza, J., Kučera, A., Mayr, R.: Quantitative analysis of probabilistic pushdown automata: Expectations and variances. In: Proceedings of LICS 2005, pp. 117–126. IEEE Computer Society Press, Los Alamitos (2005)
11. Esparza, J., Kučera, A., Schwoon, S.: Model-checking LTL with regular valuations for pushdown systems. Information and Computation 186(2), 355–376 (2003)
12. Etessami, K., Yannakakis, M.: Algorithmic verification of recursive probabilistic systems. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 253–270. Springer, Heidelberg (2005)
13. Etessami, K., Yannakakis, M.: Checking LTL properties of recursive Markov chains. In: Proceedings of 2nd Int. Conf. on Quantitative Evaluation of Systems (QEST 2005), pp. 155–165. IEEE Computer Society Press, Los Alamitos (2005)
14. Etessami, K., Yannakakis, M.: Recursive Markov chains, stochastic grammars, and monotone systems of non-linear equations. In: Diekert, V., Durand, B. (eds.) STACS 2005 [1]. LNCS, vol. 3404, pp. 340–352. Springer, Heidelberg (2005)
15. Filar, J., Vrieze, K.: Competitive Markov Decision Processes. Springer, Heidelberg (1996)
16. Grigoriev, D.: Complexity of deciding Tarski algebra. Journal of Symbolic Computation 5(1–2), 65–108 (1988)
17. Kiefer, S., Luttenberger, M., Esparza, J.: On the convergence of Newton's method for monotone systems of polynomial equations. In: Proceedings of STOC 2007, pp. 217–226. ACM Press, New York (2007)
18. Puterman, M.L.: Markov Decision Processes. Wiley, Chichester (1994)
19. Tarski, A.: A Decision Method for Elementary Algebra and Geometry. Univ. of California Press, Berkeley (1951)

A Quantifier Elimination Algorithm for Linear Real Arithmetic

David Monniaux

CNRS / VERIMAG*

Abstract. We propose a new quantifier elimination algorithm for the theory of linear real arithmetic. This algorithm uses as subroutines satisfiability modulo this theory and polyhedral projection; there are good algorithms and implementations for both of these. The quantifier elimination algorithm presented in the paper is compared, on examples arising from program analysis problems and on random examples, to several other implementations, all of which cannot solve some of the examples that our algorithm solves easily.

1 Introduction

Consider a logic formula F, possibly with quantifiers, whose variables lay within a certain set S and whose atomic predicates are relations over S. The models of this formula are assignments of values in S for the free variables of F such that F evaluates to "true". *Quantifier elimination* is the act of providing another formula F', without quantifiers, such that F and F' are *equivalent*, that is, have exactly the same models. For instance, $\forall x\ (x \geq y \Rightarrow x \geq 3)$ is equivalent to quantifier-free $y \geq 3$.

If F has no free variables, then F' is a ground (quantifier-free, variable-free) formula. In most practical cases such formulas can be easily decided to be true or false; quantifier elimination thus provides a *decision procedure* for quantified formulas.

In this paper, we only consider relations of the form $L(x, y, z, \dots) \geq 0$ where L is a linear affine expression (an arithmetic expression where multiplication is allowed only by a constant factor), interpreted over the real numbers (or, equivalently, over the rationals). We can thus deal with any formula over linear equalities or inequalities. Our algorithm transforms any formula of the form $\exists x_1, \dots, x_n\ F$, where F has no quantifiers, into a quantifier-free formula F' in disjunctive normal form. Nested quantifiers are dealt with by syntactic induction: in order to eliminate quantifiers from $\exists x\ F$ or $\forall x\ F$, where F may contain quantifiers, one first eliminates quantifiers from F. Universal quantifiers are converted to existential ones ($\forall x_1, \dots, x_n\ F \equiv \neg\exists x_1, \dots, x_n\ \neg F$), yet our algorithm generally avoids the combinatorial explosion over negations that hinders some other methods.

* VERIMAG is a joint research laboratory of CNRS and Grenoble universities.

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 243–257, 2008.

Our method can be understood as an improvement over the approach of converting to DNF through ALL-SAT and performing projection; we compared both approaches experimentally (see § 5.2). We compared our implementation with commercial and noncommercial quantifier elimination procedures over some examples arising from practical program analysis cases, as well as random problems, and ours was the only one capable of processing them without exhausting memory or time, or failing due to the impossibility of handling large coefficients.

2 The Algorithm

We first describe the datatypes on which our algorithm operates, then the off-the-shelf subroutines that it uses, then the algorithm and its correctness proof, then possible alterations.

2.1 Generalities

We process unquantified formulas built using $\wedge$, $\vee$, $\Rightarrow$, $\neg$ or other logical connectives such as exclusive-or (the exact set of connectives allowed depends on the satisfiability tester being used, see below; in this paper we shall only use $\wedge$, $\vee$ and $\neg$), and on quantified formulas built with the same connectives and the existential ($\exists$) and universal ($\forall$) quantifiers. It is possible to quantify not only on a single variable but also on a set of variables, represented as a vector $\boldsymbol{v}$. The atoms are linear inequalities, that is, formulas of the form $c + c_x x + c_y y + c_z z \cdots \geq 0$ where $c \in \mathbb{Q}$ is the *constant coefficient* and $c_v \in \mathbb{Q}$ is the coefficient associated with variable v. It is trivially possible to represent equalities or strict inequalities using this formula language. The *models* of a formula F are assignments a of rational numbers to the free variables of F such that a satisfies F (written $a \models F$). F is said to be *satisfiable* if a model exists for it. If F has no free variables, then F is said to be *true* if F is satisfiable, *false* otherwise. Two formulas A and B are said to be *equivalent*, noted $A \equiv B$, if they have the same models. Formula A is said to *imply* formula B, noted $A \Rrightarrow B$, if any model of A is a model of B.

Consider a quantifier-free formula F, whose atomic predicates are linear inequalities, and variables $x_1, \ldots, x_n$. We wish to obtain a quantifier-free formula F' equivalent to $\exists x_1, \ldots, x_n\ F$. Let us temporarily forget about efficiency in order to convince ourselves quickly that quantifier elimination is possible. F can be put into disjunctive normal form (DNF) $C_1 \vee \cdots \vee C_m$ (by recursive application of distributivity), and $\exists x_1, \ldots, x_n\ F$ is thus equivalent to $(\exists x_1, \ldots, x_n\ C_1) \vee \cdots \vee (\exists x_1, \ldots, x_n\ C_m)$. Various methods exist for finding a conjunction C'_i equivalent to $\exists x_1, \ldots, x_n\ C_i$, among which we can cite Fourier-Motzkin elimination (see § 5.1). We therefore obtain F' in DNF. For a universal quantifier, through De Morgan's laws, we obtain a formula in conjunctive normal form (CNF).

Such a naive algorithm suffers from an obvious inefficiency, particularly if applied recursively to formulas with alternating quantifiers. A first and obvious

step is to replace DNF conversion through distributivity by modern techniques (model enumeration using satisfiability modulo theory). We show in this paper than one can do better by interleaving the projection and the model numeration processes.

2.2 Building Blocks

If one has propositional formulas with a large number of variables, one never converts formulas naively from CNF to DNF, but one uses techniques such as propositional satisfiability (SAT) solving. Even though SAT is NP-complete, there now exist algorithms and implementations that can deal efficiently with many large problems arising from program verification. In our case, we apply SAT modulo the theory of linear real inequalities (SMT), a problem for which there also exist algorithms, implementations, standard benchmarks and even a competition. Like SAT, SAT modulo linear inequalities is NP-complete. A SMT solver takes as an input a formula F where the literals are linear equalities or inequalities, and answers either "not satisfiable", or a model of F, assigning a rational number to each variable in F. We assume we have such an algorithm SMT at our disposal as a building block

Another needed building block is quantifier elimination over conjunctions, named PROJECT$(C, \boldsymbol{v})$: given a conjunction C of linear inequalities over variables $\boldsymbol{v} = v_1, \ldots, v_N$, obtain a conjunction C' equivalent to $\exists v_1, \ldots, v_n\ C$. For efficiency reasons, it is better if C' is minimal (no conjunct can be removed without adding more models), or at least "small". Fourier-Motzkin elimination is a simple algorithm, yet, when it eliminates a single variable, the output conjunction can have a quadratic number of conjuncts compared to the input conjunction, thus a pass of simplification is needed for practical efficiency; various algorithms have been proposed in that respect [1]. For our implementations, we used "black box" libraries implementing geometrical transformations, in particular polyhedron projection: C defines a convex polyhedron[1] in $\mathbb{Q}^N$, and finding C' amounts to computing the inequalities defining the projection of this polyhedron into $\mathbb{Q}^{N-n}$.

3 Quantifier Elimination Algorithm

We shall first describe subroutines GENERALIZE1 and GENERALIZE2, then the main algorithm EXISTELIM.

[1] A good bibliography on convex polyhedra and the associated algorithms can be found in the documentation of the Parma Polyhedra Library. [2] By *convex polyhedron*, we mean, in a finite-dimension affine linear real space, an intersection of a finite number of half-spaces each delimited by a linear inequality, that is, the set of solutions of a finite system of linear inequalities. In particular, such a polyhedron can be unbounded. In the rest of the paper, the words "polyhedron" must be understood to mean "convex polyhedron" with that definition.

Algorithm 1. GENERALIZE1(a, F): Generalize a model a of a formula F to a conjunction

Require: $a \models F$
  $M \leftarrow \mathsf{true}$
  for all $P \in$ ATOMICPREDICATES(F) **do**
    if $a \models P$ **then**
      $M \leftarrow M \wedge P$
    else
      $M \leftarrow M \wedge \neg P$
    end if
  end for
Ensure: $M \Rrightarrow F$

Algorithm 2. GENERALIZE2(G, M): Remove useless constraints from conjunction M so that $G \wedge M \equiv \mathsf{false}$

Require: $G \wedge M$ is not satisfiable
  for all c conjunct in M **do**
    if $(G \setminus \{c\}) \wedge M$ is not satisfiable (call SMT) **then**
      remove c from M
    end if
  end for
Ensure: $G \wedge M$ is not satisfiable

3.1 Generalized Models

Consider a satisfiable quantifier-free formula F. We suppose we have at our disposal a SMT-solving algorithm that will output a model $m \models F$. We wish to obtain instead a generalized model: a conjunction C such that $C \implies F$. Ideally, we would like C to have as few conjuncts as possible. We shall now see algorithms in order to obtain such generalized models.

The truth value of F on an assignment a of its variables only depends on the truth value of the atomic predicates of F over a. Let us note $N_F = |\text{ATOMICPREDICATES}(F)|$, where $|X|$ denotes the cardinality of the set X. These truth assignments therefore define at most 2^{N_F} equivalence classes over the valuations of the variables appearing in F. There can be fewer than 2^{N_F} equivalence classes, because some truth assignments can be contradictory (for instance, $x \geq 1$ assigned to true and $x \geq 0$ assigned to false). One can immediately generalize a model of a formula to its equivalence class, which motivates our algorithm GENERALIZE1. Its output is a conjunction of literals from F.

This conjunction may itself be insufficiently general. Consider the formula $F = (x \geq 0 \wedge y \geq 0) \vee (\neg x \geq 0 \wedge y \geq 0)$. $x \mapsto 0, y \mapsto 0$ is a model of F. GENERALIZE1 will output the conjunction $x \geq 0 \wedge y \geq 0$. Yet, the first conjunct could be safely removed. GENERALIZE2$(\neg(F \vee O), M)$ will remove unnecessary conjuncts from M while preserving the property that $M \Rrightarrow F \vee O$. Figure 3 illustrates why it is better to generalize the conjunctions.

The problem of obtaining a minimal (or at least, "reasonably small") inconsistent subset out of an inconsistent conjunction has already been studied. In DPLL(T) algorithms [3] for SMT-solving, the problem is to find out, given a consistent conjunction of literals $L_1 \wedge \cdots \wedge L_n$ and a new literal L', whether $L_1 \wedge \cdots \wedge L_n \Rightarrow L'$, $L_1 \wedge \cdots \wedge L_n \Rightarrow \neg L'$, or neither; and if one of the implications holds, produce a minimal *explanation* why it holds, that is, a subset $L_{i_1}, \ldots, L_{i_m}$ of the L_i such that $L_{i_1} \wedge \cdots \wedge L_{i_m} \Rightarrow L'$ (respectively, $\Rightarrow \neg L'$). Since this decision and explanation procedure is called often, it should be fast and much effort has been devoted in that respect by implementors of SMT-solvers (e.g. [4] for congruence theories). It is however not straightforward to use such explanation procedures for our purposes, since we do not consider conjunctions of literals only: when algorithm ExistElim invokes Generalize2($\neg F, M_1$), $\neg F$ is in general a complex formula, not a literal.

We therefore present here a straightforward inconsistent set minimization algorithm similar to the one found in [5, §6]. Generalize2(G, M), where M is a conjunction such that $G \wedge M$ is unsatisfiable, works as follows:

- It attempts removing the first conjunct from M (thus relaxing the M constraint). If $G \wedge M$ stays unsatisfiable, the conjunct is removed. If it becomes satisfiable, then the conjunct is necessary and is kept.
- The process is continued with the following conjuncts.

Unsurprisingly, the results of this process depend on the order of the conjuncts inside the conjunction M. Some orders may perform better than others; the resulting set of conjuncts is minimal with respect to inclusion, but not necessarily with respect to cardinality.[2]

3.2 Main Algorithm

The main algorithm is ExistElim($F, \boldsymbol{v}$) which computes a DNF formula equivalent to $\exists \boldsymbol{v}\ F$. $\boldsymbol{v}$ is a vector of variables. $\boldsymbol{v}$ can be empty, and then the algorithm simply computes a "simple" DNF form for F. The algorithm computes generalized models of F and projects them one by one, until exhaustion. It maintains three formulas H and O. O is a DNF formula containing the projections of the models processed so far. H contains the models yet to be processed; it is initially equal to F. For each generalized model M, its projection π is added to O and removed from H. ExistElim can thus be understood as an ALL-SAT implementation coupled with a projection, where the projection is performed inside the loop so as to simplify the problem (as opposed to waiting for all models to be output and projecting them).

[2] This is the case even if we consider a purely propositional case. As an example, consider $F = A \vee (B \wedge C)$. $M = A \wedge B \wedge C \Rightarrow F$, otherwise said $M \wedge \neg F$ is not satisfiable. If one first relaxes the constraint A, one gets the conjunction $B \wedge C$, which still implies F; this conjunction has two propositional models ($A \wedge B \wedge C$ and $\neg A \wedge B \wedge C$). Yet, one could have chosen to relax B and obtain $A \wedge C$, and then to relax C and obtain A (which still implies F); this formula has four propositional models.

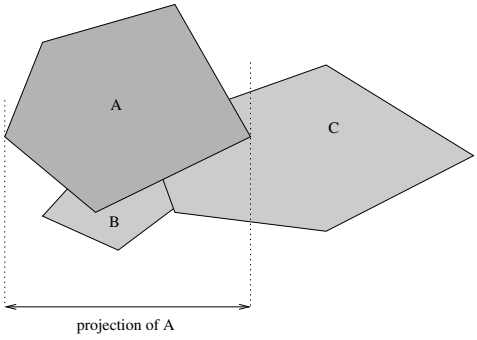


Fig. 1. Subsumption of one generalized model by another

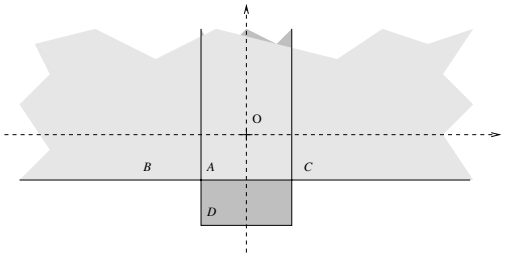


Fig. 2. The gray area is the set of points matched by formula $F = y \geq -1 \vee (y \geq -2 \wedge x \geq -1 \wedge x \leq 1)$. Point $\mathrm{O} = (0, 0)$ is found as a model. This model is first generalized to $y \geq -1 \wedge y \geq -2 \wedge x \geq -1 \wedge x \leq 1$ according to its valuations on the atomic boolean formulas. Depending on whether one first tries to relax $x \geq -1$ or $y \geq -1$, one gets either a half plane (one conjunct) or a vertical band (three conjuncts); the former is "simpler" than the second. The simplicity of the formula output by GENERALIZE2 thus depends on the ordering of the input conjuncts.

Algorithm 3. EXISTELIM: Existential quantifier elimination

$H \leftarrow F$
$O \leftarrow \mathsf{false}$
while H is satisfiable (call SMT) **do** $\{(\exists \boldsymbol{v}\ F) \equiv (O \vee \exists \boldsymbol{v}\ H)$ and $H \wedge O \equiv \mathsf{false}$ and O does not mention variables from $\boldsymbol{v}\}$
  $a \leftarrow$ a model of H $\{a \models H\}$
  $M_1 \leftarrow \textsc{Generalize1}(F, a)$ $\{M_1 \Rrightarrow F\}$
  $M_2 \leftarrow \textsc{Generalize2}(\neg F, M_1)$ $\{\neg(M_2 \wedge G)\}$
  $\pi \leftarrow \textsc{Project}(M_2, \boldsymbol{v})$ $\{\pi \equiv \exists \boldsymbol{v}\ M_2\}$
  $O \leftarrow O \vee \pi$
  $H \leftarrow H \wedge \neg\pi$
end while
Ensure: $O \equiv \exists \boldsymbol{v}\ F$

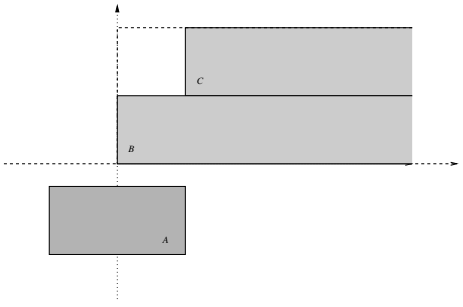


Fig. 3. A is the first generalized model selected. If $G_0 \stackrel{\text{def}}{=} \neg F$, the initial value of G, is replaced at the next iteration by $G_1 \stackrel{\text{def}}{=} \neg F \wedge \neg \pi_0$ where π_0 is the projection of A, then it is possible to generate a single generalized model encompassing both B and C (for instance $x \geq -1 \wedge y \geq 0 \wedge y \leq 2$. If G stays constant, then the $x \geq 1$ constraint defining the left edge of C cannot be relaxed.

The partial correctness of the algorithm ensues from the loop condition and the following loop invariants: $(\exists \boldsymbol{v}\ F) \equiv O \vee (\exists \boldsymbol{v}\ H)$, $H \Rrightarrow F$ and O does not mention variables from $\boldsymbol{v}$.

Given a formula ϕ, we denote by $W(\phi)$ the number of equivalence classes induced by the atomic predicates of F with nonempty intersection with the models of ϕ. Termination is ensured because $W(H)$ decreases by at least one at each iteration: M_1 defines exactly one equivalence class, M_2 defines a union of equivalence classes which includes the one defined by M_1, and the models of π include those of M_2 thus also at least one equivalence class. The number of iterations is thus at most 2^{N_F}. Note that Generalize2 is needed neither for correctness nor for termination, but only for efficiency: otherwise, the number of iterations would always be the number of equivalence classes, which can be huge.

4 Possible Changes and Extensions

We investigated two variations of the same algorithm, both of which perform significantly worse. In addition, we extended the algorithm to quantifier elimination modulo a user-specified theory.

4.1 ALL-SAT Then Project (Mod1)

The algorithm would still be correct if M was removed from H instead of π. It then becomes equivalent to performing ALL-SAT (obtaining all satisfying assignments) then projection. On the one hand, with this modified algorithm, the set of atomic formulas of H would stay included in that of F throughout the iterations, while this set can grow larger with the original algorithm since the set of atomic formulas of the projection of F can be much larger than the set

of atomic formulas in F (see §5.1). On the other hand, the original algorithm may need fewer iterations because π may subsume several generalized models, as shown by Fig. 1 : A is the first generalized model being generated, and its projection subsumes B; thus, the original algorithm will not have to generate B, while the modified algorithm will generate B. Our experiments (§5.2) showed that the unmodified algorithm often performs much better in practice than this approach.

4.2 Removals from Negated Set (Mod2)

The algorithm given previously was not the first we experimented; we had originally a slightly more complicated one, given as ExistElim(Mod2), which we wrongly thought would be more efficient. Instead of using $\neg F$ to check for inappropriate generalizations, we used a formula G initially equal to $\neg F$, and then progressively altered. The termination proof stays the same, while correctness relies on the additional invariant $G \equiv \neg(F \vee O)$. ExistElim can be thought of as identical to ExistElim(Mod2) except that G stays constant.

Algorithm 4. ExistElim(Mod2): Existential quantifier elimination

$H \leftarrow F$
$G \leftarrow \neg F$
$O \leftarrow \mathsf{false}$
while H is satisfiable (call Smt) **do** $\{(\exists \boldsymbol{v}\ F) \equiv (O \vee \exists \boldsymbol{v}\ H)$ and $G \equiv \neg(F \vee O)$ and $H \wedge O \equiv \mathsf{false}$ and O does not mention variables from $\boldsymbol{v}\}$
  $a \leftarrow$ a model of H $\{a \models H\}$
  $M_1 \leftarrow \textsc{Generalize1}(F, a)$ $\{M_1 \Rrightarrow F\}$
  $M_2 \leftarrow \textsc{Generalize2}(G, M_1)$ $\{\neg(M_2 \wedge G)\}$
  $\pi \leftarrow \textsc{Project}(M_2, \boldsymbol{v})$ $\{\pi \equiv \exists \boldsymbol{v}\ M_2\}$
  $O \leftarrow O \vee \pi$
  $H \leftarrow H \wedge \neg\pi$
  $G \leftarrow G \wedge \neg\pi$
end while
Ensure: $O \equiv \exists \boldsymbol{v}\ F$

We thought this scheme allowed more generalization of models than the algorithm we gave earlier in the article, as shown by Fig. 3. ExistElim tries to generalize M to a conjunction that implies F, but in fact this is too strict a condition to succeed, whereas ExistElim(Mod2) succeeds in generalizing F to a conjunction that implies $F \vee O$. If at least one variable is projected out, and F actually depends on that variable, then the models of F are strictly included in those of the final value of O, which is equivalent to $\exists \boldsymbol{v}\ F$.

Experiments (§5.2) however showed that this "more clever" algorithm is slower by approximately a factor of two, because adding extra assertions to G is costly for the SMT-solver.

4.3 Extra Modulo Theory

The algorithm can be easily extended to quantifier elimination modulo an assumption T on the free variables of F. All definitions stay the same except that $\Rrightarrow$ is replaced by $\Rrightarrow_T$, defined as $P \Rrightarrow_T Q \stackrel{\text{def}}{=} (P \wedge T) \Rrightarrow (Q \wedge T)$ and $\equiv$ is replaced by $\equiv_T$, defined as $(P \equiv_T Q) \stackrel{\text{def}}{=} (P \wedge T \equiv Q \wedge T)$. ExistElim is modified by replacing the initialization of G and H by $\neg F \wedge T$ and $F \wedge T$ respectively. Intuitively, T defines a universe of validity such that values outside of the models T are irrelevant to the problem being studied.

5 Comparison with Other Algorithms

The "classical" algorithm for quantifier elimination over linear inequalities is Ferrante and Rackoff's [6]. Another algorithm based on similar ideas, but with better performance, was proposed by Loos and Weispfenning [7]. We shall therefore compare our method to these algorithms, both theoretically and experimentally. We also compared our algorithm with other available packages using other quantifier elimination techniques.

5.1 Complexity Bounds

We consider in this section that inequalities are written using integer coefficients in binary notation. We shall prove that a complexity bound $2^{n^{2^q}}$ where n is the

Table 1. Timings (in seconds, on an AMD Turion TL-58 64-bit Linux system) for eliminating quantifiers from our benchmarks. The first line is the algorithm described in this paper, the two following linear variants from §4, then other packages. Reduce has `rlqe` (quantifier elimination) and `rlqe+rldnf` (same, followed by conversion to DNF). $(> t)$ means that the computation was killed after t seconds because it was running too long. The `prsb23` and following are decision problems, the output is `true` or `false`, thus DNF form does not matter. Out-of-memory is noted "o-o-m".

Benchmark	r. lim. $\mathbb{R}$	r. lim. float	`prsb23`	`blowup5`
Mjollnir	1.4	17	0.06	negligible
Mjollnir (mod1)	1.6	77[a]	0.06	negligible
Mjollnir (mod2)	1.5	34	0.07	negligible
Mjollnir Loos-Weispfenning	o-o-m	o-o-m	o-o-m	negligible
Proof-of-concept	n/a	823	n/a	n/a
Mjollnir Ferrante-Rackoff	o-o-m	o-o-m	o-o-m	negligible
Proof-of-concept	n/a	823	n/a	n/a
Lira	o-o-m	o-o-m	8.1	0.6
Redlog `rlqe`	182	o-o-m	1.4	negligible
Redlog `rlqe+rldnf`	o-o-m	o-o-m	n/a	n/a
Mathematica `Reduce`	(> 12000)	o-o-m	(> 780)	7.36

[a] Memory consumption grows to 1.1 GiB.

number of atomic formulas and q is the number of quantifiers to be eliminated. This yields an overall complexity of $2^{2^{2^{|F|}}}$ where $|F|$ is the size of the formula.

Let us consider a conjunction of inequalities taken from a set of n inequalities. The Fourier-Motzkin algorithm [8,1] eliminates variable x from this conjunction as follows. It first partitions these inequalities into those where x does not appear, which are retained verbatim, and those where x appears positively (E_+) and negatively (E_-). From each couple of inequalities (e_+, e_-) in $E_+ \times E_-$, an inequality where x does not appear is obtained by cancellation between e_+ and e_-. The size in bits of the coefficients in the output inequalities can be at most $2s + 1$ where s is the maximal size of the input coefficients.

The inequalities output therefore belong to a set of size asymptotically at most $n^2/4$ (the worst-case occurs when the inequalities split evenly between those in which x appears positively and those where it appears negatively). The output conjunction is in general too large: many inequalities in it are superfluous; yet it is guaranteed to include all inequalities defining the facets of the projection of the polyhedron.

Consider a formula F written with inequalities $A_1, \ldots, A_n$ as atomic formulas, with maximal coefficient size s. Our algorithm eliminates the quantifier from $\exists x\ F$ and outputs a DNF formula F' built with inequalities found in the output of the Fourier-Motzkin algorithm operating on the set $A_1, \ldots, A_n$ and variable x. It follows that F' is built from at most, asymptotically, $n^2/4$ inequalities as atomic formulas. The running time for this quantifier elimination comes from:

- The SMT solving passes. There are at most 2^n branches to explore in total. For each branch, SMT has to test whether the solution set of a conjunction of polynomial inequalities is empty or not, which is a particular case of linear programming, with polynomial complexity. The overall SMT cost is therefore bounded by $O(2^n.P(n))$ for some polynomial P;
- The projections, with complexity $O(n^2.s)$, applied to each of at most 2^n polyhedra.

This gives an overall complexity of $O(2^{cn})$ where c is a constant.

Consider now a succession of quantifier eliminations (with or without alternations). We now have F consisting of a sequence of quantifiers followed by a quantifier-free formula built out of atomic formulas $A_1, \ldots, A_n$. Our algorithm performs eliminations in sequence, starting from the rightmost quantifier.

Let us note $A^{(k)}$ the set of atomic formulas that can be obtained after k eliminations; $A^{(0)} = \{A_1, \ldots, A_n\}$. Clearly, $|A^{(k)}| \leq |A^{(0)}|^{2^k}$ asymptotically, since at each iteration the size of the set of atomic formulas can at most get squared by Fourier-Motzkin elimination. The size of the coefficients grows at most as $s.2^k$. This yields the promised bound.

It is possible that the bound $|A^{(k)}| \leq |A^{(0)}|^{2^k}$, obtained by observation of the Fourier-Motzkin algorithm, is too pessimistic. The literature does not show examples of such doubly exponential blowups, while polyhedra with single exponential blowups can be constructed.

The "classical" algorithm for quantifier elimination over real or rational arithmetic is Ferrante and Rackoff's method [6, 8, §7.3] [9, §4.2]. A related algorithm was proposed by Loos and Weispfenning [7] [9, §4.4]. Both these algorithms are based on the idea that an existentially quantified formula $\exists x\ F(x)$ with free variables $y, z, \ldots$ can be replaced by $F(x_1) \vee \cdots \vee F(x_m)$ where $x_1, \ldots, x_m$ are expressed as functions of $y, z, \ldots$. In the case of Ferrante and Rackoff, m is quadratic in the worst case in the length of the formula, while for Loos and Weispfenning it is linear. In both cases, the overall complexity bound is $2^{2^{cn}}$.

The weakness of both algorithms is that they never simplify formulas. This may explain that while their theoretical bounds are better than ours, our algorithm is in practice more efficient, as shown in the next subsection.

One could at first assume that the complexity bounds for our algorithm are asymptotically worse than Ferrante and Rackoff's. Our algorithm, however, outputs results in CNF or DNF form, while Ferrante and Rackoff's algorithm does not. If we add a step of transformation to CNF or DNF to their algorithm, then we also obtain a triple exponential bound.

5.2 Practical Results

We benchmarked several variants of our method against other algorithms:

Mjollnir is the algorithm described in §3, implemented on top of SMT solver YICES[3] and the NEWPOLKA polyhedron package from APRON[4], or optionally the Parma Polyhedra Library (PPL[5]). Profiling showed that most of the time is spent in the SMT solver, so performance differences between NewPolka and PPL are negligible.

Proof-of-concept is an early version of the same algorithm, implemented on top of a rudimentary SMT solver and the PPL. The SMT algorithm used is simple and lazy: the SMT problem is turned into SAT by replacing each atomic inequality by a propositional variable, and the SAT problem is input into MINISAT. A full SAT solution is obtained, then tested for emptiness by solving a linear programming problem: finding a vector of coefficients suitable as a contradiction witness for Farkas' lemma. If a witness is found, it yields a contradictory conjunction, whose negation is added to the SAT problem and SAT is restarted.

Mjollnir (mod1) is the ALL-SAT then projection algorithm from §4.1. It is invoked by option `-no-block-projected-model`.

Mjollnir (mod2) is the algorithm from §4.2; it is invoked by option `-add-blocking-to-g`.

Mjollnir Ferrante-Rackoff implements [6, 8, §7.3].

Mjollnir Loos-Weispfenning implements [7].

Lira[6] is based on Büchi automata and handles both Presburger arithmetic (integer linear inequalities) and rational linear inequalities.

[3] http://yices.csl.sri.com/

[4] http://apron.cri.ensmp.fr/library/

[5] http://www.cs.unipr.it/ppl/

[6] http://lira.gforge.avacs.org/

Table 2. Benchmarks on 3×100 random instances generated using `randprsb`, with formula depths n respectively 14, 15 and 16 (obtained by `randprsb 0 7 -10 10` n i) where i ranges in $[0, 99]$). The table shows the number of instances solved within the timeout period out of the proposed 100, the average time spent per solved instance, and the number of instances resulting in out-of-memory.

	depth 14			depth 15			depth 16		
	Solved	Avg	O-o-m	Solved	Avg	O-o-m	Solved	Avg	O-o-m
Mjollnir	100	1.6	0	94	9.8	0	73	35.3	0
Mjollnir (mod1)	94	8.2	3	80	27.3	7	39	67.1	25
Mjollnir (mod2)	100	3.8	0	91	13.9	0	65	39.2	0
Mjollnir Loos-W.	93	1.77	4	90	6.42	5	62	17.65	27
Proof-of-concept	94	1.4	0	86	2.2	0	55	17.7	0
Mjollnir Ferrante-R.	51	18.2	41	23	23.2	65	3	7.3	85
Proof-of-concept	94	1.4	0	86	2.2	0	55	17.7	0
Lira	14	102.4	83	3	77.8	94	1	8	95
Redlog (`rlqe`)	92	13.7	0	53	27.4	0	27	33.5	0
Mathematica	6	30.2	0	1	255.7	0	1	19.1	0

Mathematica[7] is a general-purpose symbolic algebra package. Its `Reduce` fonction appears to implement CAD [10], an algorithm suitable for nonlinear inequalities interpreted in the theory of real closed fields, though it is difficult to know what exactly is implemented because this program is closed source.

Redlog[8] is a symbolic formula package implemented on top of the computer algebra system Reduce 3.8.[9] Redlog implements various algorithms due to Volker Weispfenning and his group [11].

Table 1 compares these various implementations on a few benchmark examples[10] coming from two sources:

1. Examples produced from problems of program analysis following our method for the parametric computation of least invariants. [12] To summarize, each formula expresses the fact that a set of program states (such as a product of intervals for the numerical variables) is the least invariant of a program, or the strongest postcondition if there is no fixed point involved. Most of the examples, being extracted by hand from simple subprograms, were easily solved and thus did not constitute good benchmarks, but one of them, defining the least invariant of a rate limiter, proved to be tougher to solve, and we selected it as a benchmark. We have two versions of this example: the first for a rate limiter operating over real numbers ("r. lim $\mathbb{R}$") the second

[7] `http://www.wolfram.com/`
[8] `http://www.algebra.fim.uni-passau.de/~redlog/`
[9] `http://www.uni-koeln.de/REDUCE/`
[10] Available from `http://www-verimag.imag.fr/ monniaux/download/linear_qe_benchmarks.zip`

over floating-point numbers, abstracted using real numbers ("r. lim float"), and considerably tougher to process than the real example.
2. Examples procured from the LIRA designers (`prsb23` and `blowup5`).

Memory consumption stayed modest for all examples (< 15 MiB), except for r. lim float. Profiling showed that most of the time is spent in the SMT-solver and only a few percents in the projection algorithm. The fact that the proof-of-concept implementation, with a very naive SMT-solver, performs decently on an example where other algorithms exhaust memory shows that the performance of our algorithm cannot be solely explained by the good quality of YICES.

Table 2 compares the various algorithms on random examples. We then used the LIRA team's `randprsb` tool[11] to generate 100 random instances, by changing the seed of the random number generator from 0 to 99, for each of three values (14, 15, 16) of the depth parameter, which measures complexity.[12] The programs were then tested with both a 1.8 GiB memory limit and a timeout of five minutes. It is clear from Tab. 2 that `Mjollnir -no-add-blocking-to-g` is the most efficient of the tested tools.

6 Conclusion and Future Work

We have proposed a new quantifier elimination algorithm for the theory of linear inequalities over the real or rational numbers, and investigated possible variants. Our motivation was the practical application of a recent result of ours on program analysis, stating that formulas for computing the least invariants of certain kinds of systems can be obtained through quantifier elimination [12].

This algorithm is efficient on examples obtained from this program analysis technique, as well as other examples, whereas earlier published algorithms, as well as several commercial packages, all exhaust time or memory resources. Our algorithm leverages the recent progresses on satisfiability modulo theory solvers (SMT) and, contrary to older algorithms, performs on-the-fly simplifications of formulas that keep formula sizes manageable. Our algorithm also performs better than a straight application of SMT solvers (ALL-SAT followed by projection).

Our algorithm is described for rational or real linear arithmetic, but it can be extended to any theory for which there is an efficient satisfiability testing algorithm for unquantified formulas and a reasonably efficient projection algorithm for conjunctions. Among extensions that could be interesting from a practical point of view would be on the one hand the nonlinear case for real arithmetic (polynomials), and on the other hand the mixed integer / real problems. Of course, nonlinear integer arithmetic cannot be considered, since Peano arithmetic is undecidable.

[11] `http://lira.gforge.avacs.org/toolpaper/randPrsb.hs`

[12] We used the command line `randprsb 0 7 -10 10` n i where n is the depth parameter (here, 14, 15 or 16) and i ranges in $[0, 99]$.

Tarski showed that the theory of the real closed fields (inequalities of polynomial expressions) admits quantifier elimination, [13] however his algorithm had impractical (non-elementary) complexity. Later, the *cylindrical algebraic decomposition* (CAD) [14, Ch. 11] method was introduced, with doubly exponential complexity, which is unavoidable in the worst case [14, §11.4]. Our experiments with both MATHEMATICA and QEPCAD, both of which implement CAD, as well as with REDUCE/REDLOG, which implement various algorithms for quantifier elimination, showed us that combinatorial blowup occurs very quickly. For such techniques to be interesting in practice, practical complexity should be lowered. Perhaps our technique could help. There are, however, significant difficulties in that respect. Our technique starts with some single model of the target formula over the rational numbers; but a system of nonlinear inequalities needs not have rational models when it is not full-dimensional (for instance, $X^2 = 2$). Our technique reduces the geometrical computations to computations on conjunctions; but in the nonlinear case, single inequalities can be reduced to disjunctions. As an example, $X^2 \geq 4$ is reduced to $X \leq -2 \vee X \geq 2$. Most importantly, our technique relies at several steps on the availability of a decision procedure that stays efficient even when the answer is negative.

Regarding the mixed integer / real problems, the LIRA tool implements quantifier elimination using a weak form of Büchi automata matching the b-ary expression of the integers or reals, where b is an arbitrary base. [15] The output of the process is an automaton and not a readable formula. While it is possible to decide a closed formula, and to obtain one model from a satisfiable non-closed formula, it is an open problem how to efficiently reconstruct a quantifier-free formula from the resulting automaton. The automaton construct is unsuitable for large coefficients (as our examples obtained from the analysis of floating-point programs). Even on examples with small coefficients, the tool was unable to complete quantifier elimination without blowing up. We think therefore that it would be interesting to be able to apply our technique to the mixed integer / real problems, but there are difficulties: the algorithms on integer polyhedra are considerably more complex than on rational polyhedra.

A classical objection to automatic program analysis tools meant to prove the absence of bugs is that these tools could themselves contain bugs. Our method uses complex algorithms (SMT-solving, polyhedron projection) as subprocedures. We consider developing techniques so that the algorithm outputs easily-checkable proofs or "proof witnesses" of the correctness of its computation. Furthermore, we showed in earlier publications [12] that certain program analysis tasks were equivalent to quantifier elimination problems; that is, an effective static analyzer can be extracted from the quantifier-free form of an analyzer specification. This therefore suggests a new way for writing safe static analyzers: instead of painstakingly writing an analyzer, then proofs of correctness in a proof assistant [16], one could formulate the analysis problem as an equivalent quantifier elimination problem, with a relatively simple proof of equivalence, then apply a "certified" quantifier elimination procedure in order to extract the effective analyzer.

References

1. Imbert, J.L.: Fourier's elimination: Which to choose? In: Principles and Practice of Constraint Programming, pp. 117–129 (1993)
2. Bagnara, R., Hill, P.M., Zaffanella, E.: The Parma Polyhedra Library, version 0.9, http://www.cs.unipr.it/ppl
3. Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: DPLL(T): Fast Decision Procedures. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 175–188. Springer, Heidelberg (2004)
4. Nieuwenhuis, R., Oliveras, A.: Fast Congruence Closure and Extensions. Inf. Comput. 2005(4), 557–580 (2007)
5. de Moura, L., Rueß, H., Sorea, M.: Lazy theorem proving for bounded model checking over infinite domains. In: Voronkov, A. (ed.) CADE 2002. LNCS, vol. 2392, pp. 438–455. Springer, Heidelberg (2002)
6. Ferrante, J., Rackoff, C.: A decision procedure for the first order theory of real addition with order. SIAM Journal of Computation 4(1), 69–76 (1975)
7. Loos, R., Weispfenning, V.: Applying linear quantifier elimination. The Computer Journal 36(5), 450–462 (1993)
8. Bradley, A.R., Manna, Z.: The Calculus of Computation: Decision Procedures with Applications to Verification. Springer, Heidelberg (2007)
9. Nipkow, T.: Linear quantifier elimination. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS, vol. 5195, pp. 18–33. Springer, Heidelberg (2008)
10. Caviness, B.F., Johnson, J.R. (eds.): Quantifier elimination and cylindrical algebraic decomposition. Springer, Heidelberg (1998)
11. Loos, R., Weispfenning, V.: Applying linear quantifier elimination. The Computer Journal 36(5), 450–462 (1993); Special issue on computational quantifier elimination
12. Monniaux, D.: Optimal abstraction on real-valued programs. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 104–120. Springer, Heidelberg (2007)
13. Tarski, A.: A Decision Method for Elementary Algebra and Geometry. University of California Press (1951)
14. Basu, S., Pollack, R., Roy, M.F.: Algorithms in real algebraic geometry. Algorithms and computation in mathematics. Springer, Heidelberg (2003)
15. Becker, B., Dax, C., Eisinger, J., Klaedtke, F.: LIRA: handling constraints of linear arithmetics over the integers and the reals. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 312–315. Springer, Heidelberg (2007)
16. Pichardie, D.: Interprétation abstraite en logique intuitionniste : extraction d'analyseurs Java certifiés. PhD thesis, Université Rennes 1 (2005)

$\mathcal{ME}(\mathrm{LIA})$ - Model Evolution with Linear Integer Arithmetic Constraints*

Peter Baumgartner[1], Alexander Fuchs[2], and Cesare Tinelli[2]

[1] National ICT Australia (NICTA)
Peter.Baumgartner@nicta.com.au
[2] The University of Iowa, USA
{fuchs,tinelli}@cs.uiowa.edu

Abstract. Many applications of automated deduction require reasoning modulo some form of integer arithmetic. Unfortunately, theory reasoning support for the integers in current theorem provers is sometimes too weak for practical purposes. In this paper we propose a novel calculus for a large fragment of first-order logic modulo Linear Integer Arithmetic (LIA) that overcomes several limitations of existing theory reasoning approaches. The new calculus — based on the *Model Evolution* calculus, a first-order logic version of the propositional DPLL procedure — supports restricted quantifiers, requires only a decision procedure for LIA-validity instead of a complete LIA-unification procedure, and is amenable to strong redundancy criteria. We present a basic version of the calculus and prove it sound and (refutationally) complete.

1 Introduction

Many applications of automated deduction require reasoning modulo some form of integer arithmetic. Unfortunately, theory reasoning support for the integers in current theorem provers is sometimes too weak for practical purposes. We propose a novel refutation calculus for a restricted clause logic modulo Linear Integer Arithmetic (LIA) that overcomes these problems. To obtain a complete calculus, we disallow free function symbols of arity > 0 and restrict every free constant to range over a finite interval of $\mathbb{Z}$. For simplicity, we also restrict every (universal) variable to range over a bounded below interval of $\mathbb{Z}$ (such as, for instance, $\mathbb{N}$),

In spite of the restrictions, the logic is quite powerful. For instance, functions with a finite range can be easily encoded into it. This makes the logic particularly well-suited for applications that deal with bounded domains, such as, for instance, bounded model checking and planning. SAT-based techniques, based on clever reductions of BMC and planning to SAT, have achieved considerable success in the past, but they do not scale very well due to the size of the propositional formulas produced. It has been argued and shown by us and others [4,12] that this sort of applications could benefit from a reduction to a more powerful logic for which efficient decision procedures are available. That work had proposed the function-free fragment of clause logic as a candidate. This

* The work of the last two authors was partially supported by the National Science Foundation grant number 0237422.

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 258–273, 2008.

paper takes that proposal a step further by adding integer constraints to the picture. The ability to reason natively about the integers can provide a reduction in search space even for problems that do not originally contain integer constraints. The following simple example from finite model reasoning demonstrates this:[1]

$$a : [1..100] \qquad P(a) \qquad \neg P(x) \leftarrow 1 \dot{\leq} x \wedge x \dot{\leq} 100 \ .$$

The clause set above is unsatisfiable because the interval declaration $a : [1..100]$ for the constant a together with the unit clause $P(a)$ permit only models that satisfy one of $P(1), \ldots, P(100)$. Such models however falsify the third clause. Finite model finders, e.g., need about 100 steps to refute the clause set, one for each possible value of a. Our $\mathcal{ME}(\mathrm{LIA})$ calculus, on the other hand, reasons directly with integer intervals and allows a refutation in $O(1)$ steps. See Section 2 for an in-depth discussion of another example.

The calculus we propose is derived from the *Model Evolution* calculus ($\mathcal{ME}$) [7], a first-order logic version of the propositional DPLL procedure. The new calculus, $\mathcal{ME}(\mathrm{LIA})$, shares with $\mathcal{ME}$ the concept of *evolving* interpretations in search for a model for the input clause set. The crucial insight that leads from $\mathcal{ME}$ to $\mathcal{ME}(\mathrm{LIA})$ lies in the use of the ordering $<$ on integers in $\mathcal{ME}(\mathrm{LIA})$ instead of the instantiation ordering on terms in $\mathcal{ME}$. This then allows $\mathcal{ME}(\mathrm{LIA})$ to work with concepts over integers that are similar to concepts used in $\mathcal{ME}$ over free terms. For instance, it enables a strong redundancy criterion that is formulated, ultimately, as certain constraints over LIA expressions. All that requires (only) a decision procedure for the full fragment of LIA instead of a complete enumerator of LIA-unifiers.

For space constraints, we present only a basic version of the calculus. We refer the reader to a longer version of this paper [6] for extensions and improvements.

Related work. Most of the related work has been carried out in the framework of the resolution calculus. One of the earliest related calculi is theory resolution [15]. In our terminology, theory resolution requires the enumeration of a complete set of solutions of constraints. The same applies to various "theory reasoning" calculi introduced later [2,9]. In contrast, in $\mathcal{ME}(\mathrm{LIA})$ all background reasoning tasks can be reduced to *satisfiability checks* of (quantified) constraint formulas. This weaker requirement facilitates the integration of a larger class of solvers (such as quantifier elimination procedures) and leads to potentially far less calls to the background reasoner. For an extreme example, the clause $\neg(0 < x) \vee P(x)$ has, by itself, infinitely many most general LIA-unifiers (the theory reasoning analogous of most general unifiers), namely $\{x \mapsto 1\}, \{x \mapsto 2\}, \ldots$, the most general solutions of the constraint $(0 < x)$ with respect to the term instantiation ordering. Thus, any calculus based on the computation of complete sets of (most general) solutions of LIA-constraints may need to consider all of them. In contrast, in $\mathcal{ME}(\mathrm{LIA})$, or in other calculi based on *satisfiability* alone, notably Bürckert's *constrained resolution* [8], it is enough just to check that a constraint like $(0 < x)$ is LIA-satisfiable.

Constrained resolution is actually more general than $\mathcal{ME}(\mathrm{LIA})$, as it admits background theories with (infinitely, essentially enumerable) many models, as opposed to

[1] The predicate symbol $\dot{\leq}$ denotes less than or equal on integers.

the single fixed model that $\mathcal{ME}(\mathrm{LIA})$ works with.[2] On the other hand, constraint resolution does not admit free constant or function symbols—unless they are considered as part of the background theory, which is pointless since specialized background theory reasoners do not accept free symbols. The most severe drawback of constraint resolution, however, is the lack of redundancy criteria.

The importance of powerful redundancy criteria has been emphasized in the development of the modern theory of resolution in the 1990s [14]. With slight variations they carry over to *hierarchical superposition* [1], a calculus that is related to constraint resolution. The recent calculus in [11] integrates dedicated inference rules for Linear Rational Arithmetic into superposition. In [7, e.g.] we have described conceptual differences between $\mathcal{ME}$, further *instance based methods* [3] and other (resolution) calculi. Many of the differences carry over to the constraint-case, possibly after some modifications. For instance, $\mathcal{ME}(\mathrm{LIA})$ *explicitly*, like $\mathcal{ME}$, maintains a candidate model, which gives rise to a redundancy criterion different to the ones in superposition calculi. Also it is known that instance-based methods decide different fragments of first-order logic, and the same holds true for the constraint-case.

Over the last years, *Satisfiability Modulo Theories* has become a major paradigm for theorem proving modulo background theories. In one of its main approaches, DPLL(T), a DPLL-style SAT-solver is combined with a decision procedure for the quantifier-free fragment of the background theory T [13]. DPLL(T) is essentially limited to the ground case. In fact, addressing this intrinsic limitation by lifting DPLL(T) to the first-order level is one of the main motivations for the $\mathcal{ME}(\mathrm{LIA})$ calculus (much like $\mathcal{ME}$ was motivated by the goal of lifting the propositional DPLL procedure to the first-order level while preserving its good properties). At the current stage of development the core of the procedure—the Split rule—and the data structures are already lifted to the first-order level. We are working on an enhanced version with additional rules, targeting efficiency improvements. With these rules then $\mathcal{ME}(\mathrm{LIA})$ can indeed be seen as a proper lifting of DPLL(T) to the first-order level (within recursion-theoretic limitations).

2 Calculus Preview

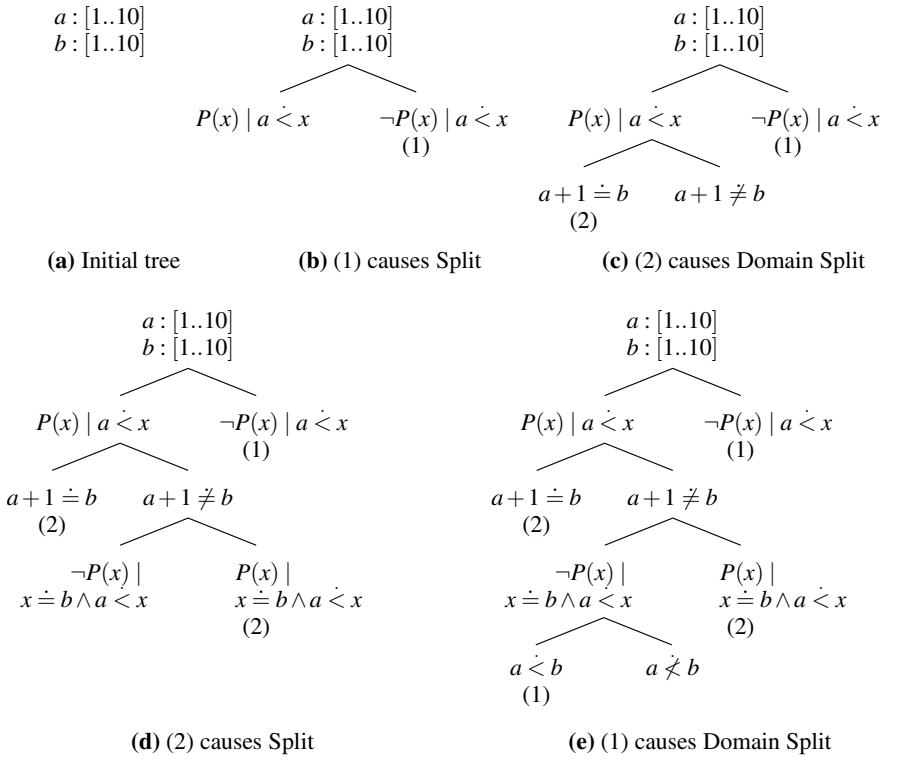
It is instructive to discuss the main ideas of the $\mathcal{ME}(\mathrm{LIA})$ calculus with a simple example before defining the calculus formally. Consider the following two unit *constrained clauses* (formally defined in Section 3):[3]

$$P(x) \leftarrow a \mathrel{\dot{<}} x \qquad (1) \qquad\qquad \neg P(x) \leftarrow x \doteq b \qquad (2)$$

where a, b are free constants, which we call *parameters*, x, y are (implicitly universally quantified) variables, and $a \mathrel{\dot{<}} x$ and $x \doteq b$ are the respective constraints of clause (1) and (2). The restriction that all parameters range over some finite integer domain is achieved with the global constraints $a : [1..10]$, $b : [1..10]$. Informally, clause (1) states that there is a value of a in $\{1, \ldots, 10\}$ such that $P(x)$ holds for all integers x greater than a. Similarly for clause (2).

[2] Extending $\mathcal{ME}(\mathrm{LIA})$ correspondingly is future work.

[3] The predicate symbol $\doteq$ denotes integer equality and $\not\doteq$ stands for $\neg(\cdot \doteq \cdot)$; similarly for $\mathrel{\dot{<}}$.



(a) Initial tree **(b)** (1) causes Split **(c)** (2) causes Domain Split

(d) (2) causes Split **(e)** (1) causes Domain Split

Fig. 1. Derivation example. Closed branches are marked with the number of the clause used to close them.

The clause set above is satisfiable in any expansion of the integers structure $\mathcal{Z}$ to $\{a,b,P\}$ that maps a,b into $\{1,\ldots,10\}$ with $a \geq b$. The calculus will discover that and compute a data structure that denotes exactly all these expansions. To see how this works, it is best to describe the calculus' main operations using a semantic tree construction, illustrated in Figure 1. Each branch in the semantic tree denotes a finite set of first-order interpretations that are expansions of $\mathcal{Z}$. These interpretations are the key to understanding the working of the calculus. The calculus' goal is to construct a branch denoting a set of interpretations that are each a model of the given clause set and the global parameter constraints, or to show that there is no such model.

In the example in Figure 1a, the initial single-node tree denotes all interpretations that interpret a and b over $\{1,\ldots,10\}$ and falsify *by default* all ground atoms of the form $P(n)$ where n is an integer constants (e.g., $P(-1), P(4), \ldots$). Each of these (100) interpretations falsifies clause (1). The calculus detects that and tries to fix the problem by changing the set of interpretations in two essentially complementary ways. It does that by computing a *context unifier* and applying the *Split* inference rule (both defined later) which extends the tree as in Figure 1b. With the addition of the *constrained literal* $P(x) \mid a \dot{\prec} x$, the left branch of the new tree now denotes all interpretations that interpret a and b as before but satisfy $P(n)$ only for values of n greater than a.

The right branch in Figure 1b still denotes the same set of interpretations as in the original branch. However, the presence of $\neg P(x) \mid a \dot{<} x$ now imposes a restriction on later extensions of the branch. To explain how, we must observe first that in the calculus the set of solutions of any constraint (which are integer tuples) is a well-founded poset. Hence, each satisfiable constraint has *minimal solutions*. Now, if a branch in the semantic tree contains a literal $L(x_1,\ldots,x_k) \mid c$ where c is a satisfiable constraint over the variables $x_1,\ldots,x_n$, each associated interpretation I satisfies $L(n_1,\ldots,n_k)$ where $(n_1,\ldots,n_k)$ is one of the minimal solutions of c in I. Further extensions of the branch must maintain $L(n_1,\ldots,n_k)$ satisfied. This minimal solution is commited to at the time the literal is added to the semantic tree. In the right branch of Figure 1b a (unique) minimal solution of $a \dot{<} x$ is $a+1$ for all interpretations. This entails that $\neg P(a+1)$ is *permanently valid* in the branch in the sense that (i) $\neg P(a+1)$ holds in every interpretation of the branch *and* (ii) no extensions of the branch are allowed to change that. As a consequence, the right branch permanently falsifies clause (1), and so it can be closed.

Similarly, $P(a+1)$ is permanently valid in the left branch of Figure 1b.[4] In interpretations of the branch where $a+1=b$ this is a problem because there clause (2) is falsified. Since the branch also has interpretations where $a+1 \neq b$, the calculus makes progress by splitting on $a+1 \doteq b$. This is done with the *Domain Split* rule, leading to the tree in Figure 1c. The leftmost branch there denotes only interpretations where $a+1=b$. That branch can be closed because it permanently falsifies clause (2). It is worth pointing out that domain splits like the above, identifying "critical" cases of parameter assignments, can be computed deterministically. They do not need not be guessed.

We skip the rest of the derivation, and just note that the trees in Figure 1d and Figure 1e are obtained by applying Split and Domain Split, respectively. As for the branch ending in $a \not\dot{<} b$, all its interpretations satisfy $P(n)$ for all $n > a$ (because the constraint in $\neg P(x) \mid x \doteq b \wedge a \dot{<} x$ is now unsatisfiable) and falsify $P(b)$ (by default, because $a \not< b$). It follows that they all satisfy the clause set. The calculus recognizes that and stops. Had the clause set been unsatisfiable, the calculus would have generated a tree with closed branches only.

Note how the calculus found a model, in fact a set of models, for the input clause set without having to enumerate all possible values for the parameters a and b, resorting instead to much more course-grained domain splits. In its full generality, the calculus still works as sketched above. Its formal description is, however, more complex because the calculus handles constraints with more than one (free) variable, and does not require the computation of explicit, symbolic representations of minimal solutions.

3 Constraints and Constrained Clauses

The new calculus works with clauses containing *parametric linear integer constraints*, which we call here simply *constraints*. These are *any first-order formulas* over the signature $\Sigma^{\Pi}_{\mathcal{Z}} = \{\doteq, \dot{<}, +, -, 0, \pm 1, \pm 2, \ldots\} \cup \Pi$, where Π is a finite set of constant symbols not in $\Sigma_{\mathcal{Z}} = \Sigma^{\Pi}_{\mathcal{Z}} \setminus \Pi$. The symbols of $\Sigma_{\mathcal{Z}}$ have the expected arity and usage. Following

[4] In DPLL terms, the split with $P(x) \mid a \dot{<} x$ and $\neg P(x) \mid a \dot{<} x$ is akin to a split on the complementary literals $P(a+1)$ and $\neg P(a+1)$. The calculus soundness proof relies in essence on this observation.

a common math terminology, we will call the elements of Π *parameters*. We will use, possibly with subscripts, the letters m,n to denote the *integer constants* (the constants in $\Sigma_{\mathcal{Z}}$); a,b to denote parameters; x,y to denote variables (chosen from an infinite set X); s,t to denote terms over $\Sigma_{\mathcal{Z}}^{\Pi}$, and l to denote literals.

We write $t : [m..n]$ as an abbreviation of $m \dot{\leq} t \wedge t \dot{\leq} n$. We denote by $\bar{\exists} c$ (resp. $\bar{\forall} c$) the existential (resp. universal) closure of the constraint c, and by $\pi_{\mathbf{x}} c$ the *projection of c on* $\mathbf{x}$, i.e., $\exists \mathbf{y}\, c$ where $\mathbf{y}$ is a tuple of all the free variables of c that are not in the variable tuple $\mathbf{x}$. We use the predicate symbol $\dot{\leq}$ also to denote the component-wise extension of the integer ordering $\dot{\leq}$ to integer tuples (for any tuple size), $\dot{\leq}_{\ell}$ to denote the lexicographic extension of $\dot{\leq}$ to integer tuples, and $\dot{<}$ and $\dot{<}_{\ell}$ to denote their respective strict version.[5]

A constraint is *ground* if it contains no variables, *closed* if it contains no free variables.[6] We define a satisfaction relation $\models_{\mathcal{Z}}$ for closed parameter-free constraints as follows: $\models_{\mathcal{Z}} c$ if c is satisfied in the standard sense in the structure $\mathcal{Z}$ of the integers—the one interpreting the symbols of $\Sigma_{\mathcal{Z}}$ in the usual way over the universe $\mathbb{Z}$. A *parameter valuation* α, a mapping from Π to $\mathbb{Z}$, determines an expansion $\mathcal{Z}_{\alpha}$ of $\mathcal{Z}$ to the signature $\Sigma_{\mathcal{Z}}^{\Pi}$ that interprets each $a \in \Pi$ as $\alpha(a)$. For each parameter valuation α and closed constraint c we write $\alpha \models_{\mathcal{Z}} c$ to denote that c is satisfied in $\mathcal{Z}_{\alpha}$. A (possibly non-closed) constraint c is *α-satisfiable* if $\alpha \models_{\mathcal{Z}} \bar{\exists} c$.

For finite sets Γ of closed constraints we denote by $\mathit{Mods}(\Gamma)$ the set of all valuations α such that $\alpha \models_{\mathcal{Z}} \Gamma$. We write $\Gamma \models_{\mathcal{Z}} c$ to denote that $\alpha \models_{\mathcal{Z}} c$ for all $\alpha \in \mathit{Mods}(\Gamma)$. For instance, $a : [1..10] \models_{\mathcal{Z}} \exists x\, x \dot{<} a$ but $a : [1..10] \not\models_{\mathcal{Z}} \exists x\, (5 \dot{<} x \wedge x \dot{<} a)$.

If e is a term or a constraint, $\mathbf{y} = (y_1, \ldots, y_k)$ is a tuple of distinct variables containing the free variables of e, and $\mathbf{t} = (t_1, \ldots, t_k)$, we denote by $e[\mathbf{t}/\mathbf{y}]$ the result of simultaneously replacing each free occurrence of y_i in e by t_i, possibly after renaming e's bound variables as needed to avoid variable capturing. We will write just $e[\mathbf{t}]$ when $\mathbf{y}$ is clear from context. With a slight abuse of notation, when $\mathbf{x}$ is a tuple of distinct variables, we will write $e[\mathbf{x}]$ to denote that the free variables of e are included in $\mathbf{x}$.

For any valuation α, a tuple $\mathbf{m}$ of integer constants is *an α-solution of* a constraint $c[\mathbf{x}]$ if $\alpha \models_{\mathcal{Z}} c[\mathbf{m}]$. For instance, $\{a \mapsto 3\} \models_{\mathcal{Z}} c[4,1]$, where $c[x,y] = (a \doteq x - y)$.

The example in the introduction demonstrated the role of minimal solutions of (satisfiable) constraints. However, minimal solutions need not always exist—consider e.g. the constraint $x \dot{<} 0$. We say that a constraint c is *admissible* iff for all parameter valuations α, if c is α-satisfiable then the set of α-solutions of c contains finitely many minimal elements with respect to $\dot{\leq}$, each of which we call a *minimal α-solution of c*. *From now on we always assume that all constraints are admissible*. Note that admissibility can be easily enforced by conjoining a given constraint $c[\mathbf{x}]$ with the constraint $\mathbf{n} \dot{\leq} \mathbf{x}$ for some tuple $\mathbf{n}$ of integer constants.

As indicated in Section 2, the calculus needs to analyse constraints and their minimal solutions. We stress that for the calculus to be effective, it need not actually *compute* minimal solutions. Instead, it is enough for it to work with constraints that *denote* each of the minimal α-solutions $m_1, \ldots, m_n$ of an α-satisfiable constraint $c[\mathbf{x}]$. This can be

[5] We remark that each of the new symbols is definable in the given constraint language.

[6] Note that a ground or closed constraint can contain parameters.

done with the formulas $\mu_k c$ defined below, where $\mathbf{y}$ is a tuple of fresh variables with the same length as $\mathbf{x}$ and $k \geq 1$.[7]

$$\mu c \stackrel{\text{def}}{=} c \wedge \forall \mathbf{y}\,(c[\mathbf{y}] \rightarrow \neg(\mathbf{y} \dot{<} \mathbf{x})) \qquad \mu_\ell c \stackrel{\text{def}}{=} c \wedge \forall \mathbf{y}\,(c[\mathbf{y}] \rightarrow \mathbf{x} \dot{\leq}_\ell \mathbf{y})$$
$$\mu_k c \stackrel{\text{def}}{=} \mu_\ell\,(\neg(\mu_1 c) \wedge \cdots \wedge \neg(\mu_{k-1} c) \wedge (\mu c))$$

Recalling that c is admissible, it is easy to see that for any valuation α, μc has at most n α-solutions (for some n): the n minimal α-solutions of c, if any. If c is α-satisfiable, let $m_1, ..., m_n$ be the enumeration of these solutions in the lexicographic order $\dot{\leq}_\ell$. Observing that $\dot{\leq}_\ell$ is a linearization of $\dot{\leq}$, it is also easy to see that $\mu_\ell c$ has exactly one α-solution: m_1. Similarly, for $k = 1, \ldots, n$, $\mu_k c$ has exactly one α-solution: m_k (this is thanks to the additional constraint $\neg(\mu_1 c) \wedge \cdots \wedge \neg(\mu_{k-1} c)$, which excludes the previous minimal α-solutions, denoted by $\mu_1 c, \ldots, \mu_{k-1}$. For $k > n$, $\mu_k c$ is never α-satisfiable. This is a formal statement of these claims:

Lemma 1. *Let α be an assignment and c an admissible constraint. Then, there is an $n \geq 0$ such that $\mu_1 c, \ldots, \mu_n c$ have unique, pairwise different α-solutions, which are all minimal α-solutions of c. Furthermore, for all $k > n$, $\mu_k c$ is not α-satisfiable.*

For example, if $c[(x,y)] = a \dot{\leq} x \wedge a \dot{\leq} y \wedge \neg(x \doteq y)$ then $\mu_\ell c$ is semantically equivalent ($\equiv$) to $x \doteq a \wedge y \doteq a+1$, $\mu c \equiv (x \doteq a \wedge y \doteq a+1) \vee (x \doteq a+1 \wedge y \doteq a)$, $\mu_1 c \equiv (x \doteq a \wedge y \doteq a+1)$, $\mu_2 c = (x \doteq a+1 \wedge y \doteq a)$ and $\mu_3 c$ is not α-satisfiable, for any α.

As we will see later, the calculus compares lexicographically minimal α-solutions of constraints that have a *single* minimal solution. With such constraints it is enough to compare their least α-solutions with respect to $\dot{\leq}_\ell$. This is done with the following comparison operators over constraints, where $\mathbf{x}$ and $\mathbf{y}$ are disjoint vectors of variables of the same length:

$$c \dot{<}_{\mu_\ell} d \stackrel{\text{def}}{=} \exists \mathbf{x} \exists \mathbf{y}\,(\mu_\ell c[\mathbf{x}] \wedge \mu_\ell d[\mathbf{y}] \wedge \mathbf{x} \dot{<}_\ell \mathbf{y}) \qquad c \doteq_{\mu_\ell} d \stackrel{\text{def}}{=} \exists \mathbf{x}\,(\mu_\ell c[\mathbf{x}] \wedge \mu_\ell d[\mathbf{x}])$$

In words, the formula $c \dot{<}_{\mu_\ell} d$ is α-satisfiable iff the least α-solutions of c and d exist, and the former is $\dot{<}_\ell$-smaller than the latter. Similarly for $c \doteq_{\mu_\ell} d$ wrt. same least α-solutions.

From the above, it is not difficult to show the following.

Lemma 2 (Total ordering). *Let α be a parameter valuation, and $c[\mathbf{x}]$ and $d[\mathbf{x}]$ two α-satisfiable (admissible) constraints. Then, exactly one of the following cases applies: (i) $\alpha \models_Z c \dot{<}_{\mu_\ell} d$, (ii) $\alpha \models_Z c \doteq_{\mu_\ell} d$, or (iii) $\alpha \models_Z d \dot{<}_{\mu_\ell} c$.*

We stress that the restriction to α-satisfiable constraints is essential here. If c or d is not α-satisfiable, then none of the listed cases applies.

3.1 Constrained Clauses

We now expand the signature Σ^{Π}_{Z} with a finite set of free predicate symbols, and denote the resulting signature by Σ. The language of our logic is made of sets of *admissible*

[7] The notations $\forall \mathbf{x}\, c$ and $\exists \mathbf{x}\, c$ stand just for c when $\mathbf{x}$ is empty.

constrained Σ*-clauses*, defined below. The semantics of the logic consists of all the expansions of the integer structure to the signature Σ, the Σ*-expansions* of $\mathcal{Z}$.

A *normalized literal* is an expression of the form $(\neg)p(\mathbf{x})$ where p is a n-ary free predicate symbol of Σ and $\mathbf{x}$ is an n-tuple of *distinct* variables. We write $L(\mathbf{x})$ to denote that L is a normalized literal whose argument tuple is *exactly* $\mathbf{x}$.

A *normalized clause* is an expression $C = L_1(\mathbf{x}_1) \vee \cdots \vee L_n(\mathbf{x}_n)$ where $n \geq 0$ and each $L_i(\mathbf{x}_i)$ is a normalized literal, called a *literal in* C. We write $C(\mathbf{x})$ to indicate that C is a normalized clause whose variables are exactly $\mathbf{x}$. We denote the empty clause by $\Box$.

A *(constrained* Σ*-)clause* $D[\mathbf{x}]$ is an expression of the form $C(\mathbf{x}) \leftarrow c$ with the free variables of c included in $\mathbf{x}$. When C is $\Box$ we call D a *constrained empty clause*. A clause $C(\mathbf{x}) \leftarrow c$ is *LIA-(un)satisfiable* if there is an (no) Σ-expansion of the integer structure $\mathcal{Z}$ that satisfies $\forall \mathbf{x}\,(c \rightarrow C(\mathbf{x}))$. A set S of clauses and constraints is *LIA-(un)satisfiable* if there is an (no) Σ-expansion of $\mathcal{Z}$ that satisfies every element of S.

We will consider only *admissible clauses*, i.e., constrained clauses $C(\mathbf{x}) \leftarrow c$ where (i) $C \neq \Box$ and (ii) c is an admissible constraint. Condition (i) above is motivated by purely technical reasons. It is, however, no real restriction, as any clause $\Box \leftarrow c$ in a clause set S can be replaced by $\mathit{false} \leftarrow c$, where *false* is a 0-ary predicate symbol not in S, once S has been extended with the clause $\neg \mathit{false} \leftarrow \top$.[8] Condition (ii) is the real restriction, needed to guarantee the existence of minimal solutions, as explained earlier. To simplify the presentation, we will further restrict ourselves to clauses with (trivially admissible) constrains of the form $c[\mathbf{x}] \wedge \mathbf{0} \dot{\leq} \mathbf{x}$, where $\mathbf{0}$ is the tuple of all zeros. For brevity, in our examples we will sometimes leave the constraint $\mathbf{0} \dot{\leq} \mathbf{x}$ implicit.

4 Constrained Contexts

A *context literal* K is a pair $L(\mathbf{x}) \mid c$ where $L(\mathbf{x})$ is a normalized literal and c is an (admissible) constraint with free variables included in $\mathbf{x}$. We denote by $\overline{K}$ the constrained literal $\overline{L}(\mathbf{x}) \mid c$, where $\overline{L}$ is the complement of L.

A *(constrained) context* is a pair $\Lambda \cdot \Gamma$ where Γ is a finite set of closed constraints and Λ is a finite set of context literals. We will implicitly identify the sets Λ with their closure under renamings of a context literal's free variables.

In terms of the semantic tree presentation in Figure 1, each branch there corresponds (modulo a detail explained below) to a context $\Lambda \cdot \Gamma$, where Γ are the parameter constraints along the branch and Λ are the constrained literals. In the discussion of Figure 1 we explained informally the meaning of parameter constraints and constrained literals. The purpose of this section is to provide a formal account for that.

Definition 3 **(α-Covers, α-Extends).** *Let* α *be a parameter valuation. A context literal* $L(\mathbf{x}) \mid c_1$ α*-covers a context literal* $L(\mathbf{x}) \mid c_2$ *if* $\alpha \models_{\mathcal{Z}} \bar{\exists}\, c_2$ *and* $\alpha \models_{\mathcal{Z}} \bar{\forall}\, (c_2 \rightarrow c_1)$.

The literal $L(\mathbf{x}) \mid c_1$ α*-extends* $L(\mathbf{x}) \mid c_2$ *if* $L(\mathbf{x}) \mid c_1$ α*-covers* $L(\mathbf{x}) \mid c_2$ *and* $\alpha \models_{\mathcal{Z}} c_1 \doteq_{\mu_\ell} c_2$. *If* Γ *is a set of closed constraints,* $L(\mathbf{x}) \mid c_1$ Γ*-extends* $L(\mathbf{x}) \mid c_2$ *if it* α*-extends it* for all $\alpha \in \mathit{Mods}(\Gamma)$.

[8] We will use $\top$ and $\bot$ respectively for the universally true and the universally false constraint.

For an unnormalized literal $L(\mathbf{t})$ we say that $L(\mathbf{x}) \mid c_1[\mathbf{x}]$ α-*covers* $L(\mathbf{t})$ if $L(\mathbf{x})$ covers the normalized version of $L(\mathbf{t})$, i.e., the literal $L(\mathbf{x}) \mid \pi\,\mathbf{x}\,(\mathbf{x} \doteq \mathbf{t}[\mathbf{z}/\mathbf{x}])$ where $\mathbf{z}$ is a tuple of fresh variables.

The intention of the previous definition is to compare context literals with respect to their set of solutions for a fixed valuation α. This is expressed basically by the second condition in the definition of α-covers. For example, $P(x) \mid a \dot{<} x$ α-covers $P(x) \mid a+1 \dot{<} x$, for any α. The first condition ($\alpha \models_{\mathcal{Z}} \bar{\exists}\, c_2$) is needed to exclude α-coverage for trivial reasons, because c_2 is not α-satisfiable. Without it, for example, $P(x) \mid x \doteq 2$ would α-cover $P(x) \mid x \doteq a \wedge a \doteq 5$ when, say, $\alpha(a) = 3$, which is not intended. But note that $\alpha \not\models_{\mathcal{Z}} \exists x\,(x \doteq a \wedge a \doteq 5)$ in this case. Also note that the two conditions $\alpha \models_{\mathcal{Z}} \bar{\exists}\, c_2$ and $\alpha \models_{\mathcal{Z}} \bar{\forall}\,(c_2 \to c_1)$ in combination enforce that c_1 is α-satisfiable as well.

The notion of α-extension is similar to that of α-coverage, but applies to literals with the same least solutions only. For instance, $P(x) \mid 0 \dot{\leq} x \wedge x \dot{<} 7$ α-extends $P(x) \mid 0 \dot{\leq} x \wedge x \dot{<} 3$, and α-covers it, for any α (the least solution being 0 for both literals), and $P(x) \mid 3 \dot{<} x$ α-covers $P(x) \mid 7 \dot{<} x$ but does not α-extend it.

The concepts introduced in the next three definitions allow us to associate a set of structures to each context satisfying certain well-formedness conditions.

Definition 4 (α-Produces). *Let Λ be a set of constrained literals and α a parameter valuation. A context literal $L(\mathbf{x}) \mid c_1$ α-produces a context literal $L(\mathbf{x}) \mid c_2$ wrt. Λ if*

1. *$L(\mathbf{x}) \mid c_1$ α-covers $L(\mathbf{x}) \mid c_2$, and*
2. *there is no $\overline{L}(\mathbf{x}) \mid d$ in Λ that α-covers $\overline{L}(\mathbf{x}) \mid c_2$ and such that $\alpha \models_{\mathcal{Z}} c_1 \dot{<}_{\mu_\ell} d$.*

The set Λ α-produces a context literal K if some literal in Λ α-produces K wrt. Λ. A context $\Lambda \cdot \Gamma$ produces K if there is an $\alpha \in \mathit{Mods}(\Gamma)$ such that Λ α-produces K.

As an example, if $\alpha(a) = 3$ then $P(x) \mid 2 \dot{<} x$ α-produces $P(5)$ wrt. $\Lambda = \{\neg P(x) \mid x \doteq a \wedge a \doteq 5\}$. Observe that neither $\alpha \models_{\mathcal{Z}} (2 \dot{<} x) \dot{<}_{\mu_\ell} (x \doteq a \wedge a \doteq 5)$ holds nor does $\neg P(x) \mid x \doteq a \wedge a \doteq 5$ α-cover $\neg P(5)$, as $x \doteq a \wedge a \doteq 5$ is not α-satisfiable. However, if $\alpha(a) = 5$ then $P(x) \mid 2 \dot{<} x$ no longer α-produces $P(5)$ wrt. Λ, because now $\alpha \models_{\mathcal{Z}} (2 \dot{<} x) \dot{<}_{\mu_\ell} (x \doteq a \wedge a \doteq 5)$ and $\neg P(x) \mid x \doteq a \wedge a \doteq 5$ α-covers $\neg P(5)$.

Definition 5 (α-Contradictory). *Let $\Lambda \cdot \Gamma$ be a context and $\alpha \in \mathit{Mods}(\Gamma)$. A context literal $L(\mathbf{x}) \mid c$ is α-contradictory with Λ if there is a context literal $\overline{L}(\mathbf{x}) \mid d$ in Λ such that $\alpha \models_{\mathcal{Z}} c \doteq_{\mu_\ell} d$. It is Γ-contradictory with Λ if there is a $\overline{L}(\mathbf{x}) \mid d$ in Λ such that $\Gamma \models_{\mathcal{Z}} c \doteq_{\mu_\ell} d$.*

The literal $L(\mathbf{x}) \mid c$ is contradictory with the context *$\Lambda \cdot \Gamma$ if it is α-contradictory with Λ for some $\alpha \in \mathit{Mods}(\Gamma)$. The context $\Lambda \cdot \Gamma$ itself is* contradictory *if some context literal in Λ is contradictory with it.*

The notion of Γ-contradictory is based on equality of the least α-solutions of the involved constraints for all $\alpha \in \mathit{Mods}(\Gamma)$. It underlies the abandoning of candidate models due to permanently falsified clauses in Section 2, which is captured precisely as *closing literals* in Definition 8 below.

We require our contexts not only to be non-contradictory but also to constrain each parameter to a finite subset of $\mathbb{Z}$. Furthermore, they should guarantee that the associated Σ-expansions of $\mathcal{Z}$ are total over tuples of natural numbers. All this is achieved with *admissible* contexts.

Definition 6 (Admissible Γ, Admissible Context). *A context $\Gamma \cdot \Lambda$ is admissible if*

1. *Γ is admissible, that is, Γ is satisfiable, and, for each parameter a in Π, there are integer constants $m, n \geq 0$ such that $\Gamma \models a : [m..n]$.*
2. *For each free predicate symbol P in Σ, the set Λ contains $\neg P(\mathbf{x}) \mid -\mathbf{1} \dot{\leq} \mathbf{x}$.*
3. *$\Lambda \cdot \Gamma$ is not contradictory.*[9]

Thanks to Condition 2 in the above definition, an admissible context α-produces a literal $\neg P(\mathbf{n})$ with $\mathbf{n}$ consisting of non-negative integer constants, if no other literal in the context α-produces $P(\mathbf{n})$. Observe that admissible contexts $\Lambda \cdot \Gamma$ may contain context literals whose constraint is not α-satisfiable for some (or even all) $\alpha \in \mathit{Mods}(\Gamma)$. For those α's, such literals simply do not matter as their effect is null.

However, admissible contexts are always consistent in the sense that they cannot produce both a constraint literal $L(\mathbf{x}) \mid c$ and its complement $\overline{L}(\mathbf{x}) \mid c$. The following definition provides the formal account of the meaning of contexts announced at the beginning of this section.

Definition 7 (Induced Structure). *Let $\Gamma \cdot \Lambda$ be an admissible context and let $\alpha \in \mathit{Mods}(\Gamma)$. The Σ-structure $\mathcal{Z}_{\Lambda,\alpha}$ induced by Λ and α is the expansion of $\mathcal{Z}$ to all the symbols in Σ that interprets each parameter a as $\alpha(a)$, and satisfies a positive ground literal $L(\mathbf{s})$ iff Λ α-produces $L(\mathbf{s})$.*

The above consistency property and the presence of literals $\neg P(\mathbf{x}) \mid -\mathbf{1} \dot{\leq} \mathbf{x}$ in admissible contexts entails that, for every $\alpha \in \mathit{Mods}(\Gamma)$, $\mathcal{Z}_{\Lambda,\alpha}$ satisfies a literal $L(\mathbf{n})$ if and only if Λ α-produces $L(\mathbf{n})$, where $\mathbf{n}$ is a tuple of non-negative integer constants. Thus, Definition 7 connects syntax (α-productivity) to semantics (truth) in a one-to-one way.

In Section 2 we explained the derivation in Figure 1 as being driven by semantic considerations, to construct a model by successive branch extensions. The calculus' inference rules achieve that in their core by computing *context unifiers*.

Definition 8 (Context Unifier). *Let $\Lambda \cdot \Gamma$ be an admissible context and $D[\mathbf{x}] = L_1(\mathbf{x}_1) \vee \cdots \vee L_k(\mathbf{x}_k) \leftarrow c[\mathbf{x}]$ a constrained clause with free variables $\mathbf{x}$. A context unifier of D against $\Lambda \cdot \Gamma$ is a constraint*

$$d[\mathbf{x}] = d'[\mathbf{x}] \wedge \exists \mathbf{y}\,(\mathbf{y} \dot{\leq} \mathbf{x} \wedge \mu_j\, d'[\mathbf{y}]), \quad \text{where } d'[\mathbf{x}] = c[\mathbf{x}] \wedge c_1[\mathbf{x}_1] \wedge \cdots \wedge c_k[\mathbf{x}_k] \qquad (1)$$

with each c_i coming from a literal $\overline{L_i}(\mathbf{x}_i) \mid c_i$ in Λ, and $j \geq 1$.

For each $i = 1, \ldots, k$, the context literal

$$L_i(\mathbf{x}_i) \mid d_i, \text{ with } d_i = \pi\, \mathbf{x}_i\, d \qquad (2)$$

is a literal of *the context unifier. The literal $L_i(\mathbf{x}_i) \mid d_i$ is closing if $\Gamma \models_{\mathcal{Z}} c_i \doteq_{\mu_\ell} d_i$. Otherwise, it is a ($\alpha$-)*remainder literal *(of d) if there is an $\alpha \in \mathit{Mods}(\Gamma)$ such that $\alpha \models_{\mathcal{Z}} c_i <_{\mu_\ell} d_i$ (equivalently, such that $\alpha \not\models_{\mathcal{Z}} c_i \doteq_{\mu_\ell} d_i$ and d_i is α-satisfiable)*[10].

The context unifier d is closing *if each of its literals is closing. It is (α-)*productive *if for each $i = 1, \ldots, k$, the context literal $\overline{L_i}(\mathbf{x}_i) \mid c_i$ α-produces $\overline{L_i}(\mathbf{x}_i) \mid d_i$ wrt. Λ for some $\alpha \in \mathit{Mods}(\Gamma)$.*

[9] Equivalently, for every $\alpha \in \mathit{Mods}(\Gamma)$ and every pair of context literals $L(\mathbf{x}) \mid c$ and $\overline{L}(\mathbf{x}) \mid d$ in Λ, it is *not* the case that $\alpha \models_{\mathcal{Z}} c \doteq_{\mu_\ell} d$.

[10] Observe that if d_i is α-satisfiable so are d and c_i.

The constraint d in (1) can be perhaps best understood as follows. Its component $d' = c[\mathbf{x}] \wedge c_1[\mathbf{x}_1] \wedge \cdots \wedge c_k[\mathbf{x}_k]$ denotes any simultaneous solution of D's constraint and the constraints coming from pairing each of D's literal with a context literal with same predicate symbol but opposite sign. The component $\mu_j d'[\mathbf{y}]$ denotes the j^{th} minimal solution of d', which bounds from below the solutions of d. A simple, but important consequence (for completeness) is that for any α and concrete solution $\mathbf{m}$ of d', j can be always chosen so that $d[\mathbf{m}]$ is α-satisfied. As a special case, when $\mathbf{m}$ is the j-th minimal solution of d', it is also the least solution of d. Regarding d_i in (2), for any α, the set of α-solutions of d_i is the projection over the vector $\mathbf{x}_i$ of the solutions of d.

A formal statement of the above is expressed by the following lemma.

Lemma 9 (Lifting). *Let $\Lambda \cdot \Gamma$ be an admissible context, $\alpha \in Mods(\Gamma)$, $D[\mathbf{x}] = L_1(\mathbf{x}_1) \vee \cdots \vee L_k(\mathbf{x}_k) \leftarrow c[\mathbf{x}]$ with $k \geq 1$ a constrained clause, and $\mathbf{m}$ a vector of constants from $\mathbb{Z}$. If $\mathcal{Z}_{\Lambda,\alpha}$ falsifies $D[\mathbf{m}]$, then there is an α-productive context unifier d of D against $\Lambda \cdot \Gamma$ where $\mathbf{m}$ is an α-solution of d.*

As an example (with no parameters, for simplicity), let $d' = c[x_1,x_2] \wedge c_1[x_1] \wedge c_2[x_2]$ where $c = \neg(x_1 \doteq x_2)$, $c_1 = 1 \dot{\leq} x_1$, and $c_2 = 1 \dot{\leq} x_2$. Then, the (unique) solution of $\mu_j d'$ for $j = 1$ is $(1,2)$; for $j = 2$ it is $(2,1)$. By fixing $j = 1$ now let us commit to $(1,2)$. Then the solutions of d_1 are $(1),(2),\ldots$ and the solutions of d_2 are $(2),(3),\ldots$ The least solution of d_1, (1), coincides with the projection over x_1 of the commited minimal solution $(1,2)$. Similarly for d_2. This is no accident and is crucial in proving the soundness of the calculus. It relies on the property that the least (individual) solutions of all the d_i's are, in combination, the least solution of d—which is in turn the first minimal solution of d'. In the example, the least solutions of d_1 and d_2 are 1 and 2, respectively, and combine into $(1,2)$, the least solution of d.

We stress that all the notions in the above definition are effective thanks to the decidability of LIA. A subtle point here is the choice of j in (1), as j is not bounded *a priori*. However, all these notions hold only if d_i is α-satisfiable for some or all (finitely) many choices of $\alpha \in Mods(\Gamma)$, and that d_i becomes α-unsatisfiable if j exceeds the number of minimal α-solutions of d_i. By this argument, the possible values for j are effectively bounded.

Example 10. Consider the context $\{P(x) \mid a \dot{<} x\} \cdot \{a : [1..10], b : [1..10]\}$ and the input clause $\neg P(x) \leftarrow b \dot{<} x$. The context corresponds to the left branch in Figure 1b. There is a context unifier, for any $j \geq 1$, $d = a \dot{<} x \wedge b \dot{<} x \wedge \exists y\,(y \dot{\leq} x \wedge \mu_j(a \dot{<} y \wedge b \dot{<} y))$. Its literal is $K' = \neg P(x) \mid d_1$, where $d_1 = \pi x d (= d)$. The constraint $(a \dot{<} y \wedge b \dot{<} y)$ has a unique minimal α-solution, which is also its least α-solution. Thus, d is equivalent to $a \dot{<} x \wedge b \dot{<} x$, obtained with $j = 1$. It is closing if $\Gamma \models_{\mathcal{Z}} (a \dot{<} x) \doteq_{\mu_\ell} d_1$, which is equivalent to $\Gamma \models_{\mathcal{Z}} \neg(a \dot{<} b)$. That is not the case, i.e. there is an $\alpha \in Mods(\Gamma)$ that satisfies $a \dot{<} b$. According to Definition 8 then, K' is a remainder literal of d. Indeed, it can be verified then that $\alpha \models_{\mathcal{Z}} (a \dot{<} x) \dot{<}_{\mu_\ell} (a \dot{<} x \wedge b \dot{<} x)$.

5 The Calculus

The inference rules of the calculus are defined over triples, *sequents*, of the form $\Lambda \cdot \Gamma \vdash \Phi$ where $\Lambda \cdot \Gamma$ is an admissible context and Φ is a set of constrained clauses. Intuitively,

an antecedent $\Lambda \cdot \Gamma$ corresponds to a branch in the semantic tree presentation in Section 2 and always denotes *a set* of candidate models for Φ, the Σ-structures induced by Λ and $\alpha \in \mathit{Mods}(\Gamma)$ (Def. 7) .

The calculus derives a tree of sequents with the goal of incrementally modifying the candidate models until they *evolve*, so to speak, into a set of models of Φ. More precisely, a *derivation* of Γ and Φ starts with a tree with a root node only, which is labeled with the sequent $\Lambda_0 \cdot \Gamma \vdash \Phi$, where Λ_0 contains (only) the constraint literal $\neg p(\mathbf{x}) \mid -\mathbf{1} \doteq \mathbf{x}$ for each free predicate symbol p in Σ. It then applies the derivation rules defined below to grow that tree, by applying a rule at a time to a leaf of the tree and extending it with the conclusions in the expected way. See [6] for a formal definition.

Context unifiers play a crucial role in the evolution of $\Lambda \cdot \Gamma$. To illustrate their use, consider a sequent $\Lambda \cdot \Gamma \vdash \Phi$. If for some $\alpha \in \mathit{Mods}(\Gamma)$ the structure $\mathcal{Z}_{\Lambda,\alpha}$ induced by Λ and α falsifies Φ, it must falsify a "ground" instance $D[\mathbf{m}]$ of some clause D in Φ. As shown in [6], this implies the existence of an α-productive context unifier d of D against $\Lambda \cdot \Gamma$ where $\mathbf{m}$ is an α-solution of d.

If d has an α-remainder literal $K'_i = L(\mathbf{x}_i) \mid d_i$ not contradictory with the context, the problem with $D[\mathbf{m}]$ can be fixed by adding K'_i to Λ. In fact, if $\mathbf{m}_i$ is the projection of $\mathbf{m}$ over $\mathbf{x}_i$, then K'_i will α-produce $L_i(\mathbf{m}_i)$ in the new context, as its least solution is no greater than $\mathbf{m}_i$.[11] That will make the new $\mathcal{Z}_{\Lambda,\alpha}$ satisfy $L_i(\mathbf{m}_i)$ and so $D[\mathbf{m}]$ as well. This is essentially what the calculus does to $\Lambda \cdot \Gamma \vdash \Phi$ with the rules $\mathsf{Split}(d)$ or $\mathsf{Extend}(d)$ introduced below. If each α-remainder literal of d is contradictory with the context, it will be β-contradictory with Λ for one or more $\beta \in \mathit{Mods}(\Gamma)$. Then, it is necessary to strengthen Γ to eliminate the offending β's, which is achieved with the $\mathsf{Domain\ Split}(d)$ rule. Strengthening Γ either makes $\mathsf{Split}(d)$ or $\mathsf{Extend}(d)$ applicable to an α-remainder literal of d or turns all literals of d into closing ones. In the latter case, the calculus will close the corresponding branch with the $\mathsf{Close}(d)$ rule.

The $\mathcal{ME}$(LIA) calculus has four derivation rules. The application of these rules is subject to certain fairness criteria, explained later. In the rules, the notation Φ, D abbreviates $\Phi \cup \{D\}$. (Similarly for Λ, K and Γ, c.)

$$\mathsf{Close}(d)\quad \frac{\Lambda \cdot \Gamma \vdash \Phi, D}{\Lambda \cdot \Gamma \vdash \Phi, D, \square \leftarrow \top} \text{ if } \begin{cases} (\square \leftarrow \top) \notin \Phi \cup \{D\}, \text{ and} \\ d \text{ is a closing context unifier of } D \text{ against } \Lambda \cdot \Gamma. \end{cases}$$

This rule recognizes that the context not only falsifies some input clause D but is also unfixable, and adds the empty clause as a marker for that.

$$\mathsf{Split}(d)\quad \frac{\Lambda \cdot \Gamma \vdash \Phi, D}{(\Lambda, L_i \mid d_i) \cdot \Gamma \vdash \Phi, D \qquad (\Lambda, \overline{L_i} \mid d_i) \cdot \Gamma \vdash \Phi, D} \text{ if } \begin{cases} d \text{ is a context unifier of } D \text{ against } \Lambda \cdot \Gamma, \\ L_i \mid d_i \text{ is a remainder literal of } d, \text{ and} \\ \text{neither } L_i \mid d_i \text{ nor } \overline{L_i} \mid d_i \text{ is contradictory} \\ \qquad \text{with } \Lambda \cdot \Gamma. \end{cases}$$

This rule, analogous to the main rule of the DPLL procedure, derives one of two possible sequents non-deterministically. The left-hand side conclusion chooses to fix the context by adding $L_i \mid d_i$ to Λ. The right-hand side branch is needed for soundness, in case the left-hand side fix leads to an application of Close. It causes progress in the

[11] This is the analogous of "lifting" in a Herbrand-based theorem proving.

derivation by making $L_i \mid d_i$ Γ-contradictory with the context, which forces the calculus to consider other alternatives to $L_i \mid d_i$.

$$\textsf{Extend}(d)\quad \frac{\Lambda \cdot \Gamma \qquad \vdash \Phi, D}{(\Lambda, L_i \mid d_i) \cdot \Gamma \vdash \Phi, D} \text{ if } \begin{cases} d \text{ is a context unifier of } D \text{ against } \Lambda \cdot \Gamma, \\ L_i \mid d_i \text{ is a remainder literal of } d, \\ \overline{L_i} \mid d_i \text{ is } \Gamma\text{-contradictory with } \Lambda, \text{ and} \\ \text{there is no } K \text{ in } \Lambda \text{ that } \Gamma\text{-extends } L_i \mid d_i. \end{cases}$$

This rule can be seen as a one-branched Split. If $\overline{L_i} \mid d_i$ is Γ-contradictory with Λ, the only way to fix the context is to add $L_i \mid d_i$ to it. Its last precondition is a redundancy test—which also prevents a repeated application of the rule with the same literal.

To illustrate the need of Extend, suppose $\Lambda = \{\neg P(x) \mid -1 \dot{\leq} x,\ P(x) \mid x : [1\,..\,5]\}$, $\Gamma = \emptyset$ and $D = P(x) \leftarrow x : [1\,..\,7]$. The clause D is falsified in the (single) induced interpretation[12]. Adding $P(x) \mid x : [1\,..\,7]$ to Λ will fix the problem. However, Split cannot be used for that since $\neg P(x) \mid x : [1\,..\,7]$ is Γ-contradictory with Λ—for having the same least solution, 1, as the constraint of $P(x) \mid x : [1\,..\,5]$. Extend will do instead.

$$\textsf{Domain Split}(d)\quad \frac{\Lambda \cdot \Gamma \ \vdash\ \Phi, D}{\Lambda \cdot (\Gamma, c \doteq_{\mu_\ell} d_i) \ \vdash\ \Phi, D \qquad \Lambda \cdot (\Gamma, \neg(c \doteq_{\mu_\ell} d_i)) \ \vdash\ \Phi, D} \text{ if } \begin{cases} d \text{ is a c.u. of } D \text{ against } \Lambda \cdot \Gamma, \\ \text{there is a literal } L_i \mid d_i \text{ of } d, \text{ and} \\ \text{there is } \overline{L_i} \mid c \text{ or } L_i \mid c \text{ in } \Lambda \text{ s.t.} \\ \quad \alpha \models_{\mathcal{Z}} c \doteq_{\mu_\ell} d_i \\ \quad \text{for some } \alpha \in \mathit{Mods}(\Gamma), \text{ and} \\ \quad \Gamma \not\models_{\mathcal{Z}} c \doteq_{\mu_\ell} d_i. \end{cases}$$

The purpose of this rule is to enable later applications of the other rules that are not applicable to the current context. It does that by partitioning the current $\mathit{Mods}(\Gamma)$ in two non-empty parts.

It is not too difficult to see that the derivation rules are mutually exclusive, in the sense that for a given sequent at most one of them is applicable to the same clause D, context unifier d, and literal of d.

In [6] we introduce another optional rule, Ground Split, that adds another, more flexible, way to do case analysis on the parameters. The rule can improve efficiency in particular when paired with a suitable quantifier elimination procedure for LIA. In that case, one can replace each potential application of Domain Split, which would add a constraint $[\neg](c \doteq_{\mu_\ell} d_i)$ to Γ, with one application of Ground Split, which splits on a ground constraint l that entails $c \doteq_{\mu_\ell} d_i$ and is computed from it by the QE procedure. The net effect is that Γ grows only with ground literals, making tests involving it considerably cheaper—at the cost of an increased number of splits for Γ.

5.1 Soundness and Completeness

Proposition 11 (Soundness). *For all admissible clause sets Φ and admissible sets of closed constraints Γ, if there is a derivation of Φ and Γ that ends in a tree containing $\square \leftarrow \top$ in each of its leaf nodes, then $\Gamma \cup \Phi$ is LIA-unsatisfiable.*

In essence, and leaving Γ aside, the proof is by first deriving a binary tree over ground, parameter-free literals that reflects the applications of the derivation rules in the

[12] Because, for instance, $\neg P(6)$ is true in it.

construction of the given refutation tree. For instance, a Split application with its new constraint literal $L(\mathbf{x}) \mid c$ in the left context gives rise to the literal $L(\mathbf{m})$, where $\mathbf{m}$ is the least α-solution of c for a given α. In the resulting tree neighbouring nodes will be labelled with complementary literals, like $L(\mathbf{m})$ and $\neg L(\mathbf{m})$. In the second step it is shown that this binary tree is closed by ground instances from the input set. It is straightforward then to argue that $\Phi \cup \Gamma$ is LIA-unsatisfiable.

To prove the calculus' completeness requires to introduce several technical notions. Again we refer to the long version of this paper [6] for that, and provide a brief summary here only. One of these notions is that of an *exhausted branch*, in essence, a (limit) derivation tree branch that need not be extended any further. It is based on the notion of *redundant context unifiers*.

Definition 12 (Redundant Context Unifier). *Let $\Lambda_1 \cdot \Gamma_1$ and $\Lambda_2 \cdot \Gamma_2$ be admissible contexts, $\alpha \in Mods(\Gamma_1)$ and D a clause. A context unifier d of D against $\Lambda_1 \cdot \Gamma_1$ is α-redundant in $\Lambda_2 \cdot \Gamma_2$ if*

1. *Λ_2 α-produces some literal of d, or*
2. *$Mods(\Gamma_2) \subsetneq Mods(\Gamma_1)$*

We say that d is redundant *in $\Lambda_2 \cdot \Gamma_2$ if it is α-redundant in $\Lambda_2 \cdot \Gamma_2$ for all $\alpha \in Mods(\Gamma)$.*

If condition (1) applies then the interpretation induced by Λ_2 and α will already satisfy D, and there is no point considering a derivation rule application based on that d. Condition (2) allows us to discard an existing derivation rule application when the constraints in Γ are strengthened.

Now, an *exhausted (limit) branch* (i) has the property that whenever Split, Extend or Domain Split is applicable to some of its sequents, based on an α-productive context unifier, then this context unifier is α-redundant in the context of some later sequent (a sequent more distant from the root), (ii) cannot be applied Close to, and (iii) does not contain $\square \leftarrow \top$. Finally, in *fair derivations* each leaf node of some derived tree contains $\square \leftarrow \top$ or its limit tree has an exhausted branch.

Fair derivations in the sense above exist and are computable for any set of Σ-clauses. A naive fair proof procedure, for instance, grows a branch until the above conditions (ii) and (iii) are violated, and turns to another branch to work on, if any, or otherwise applies the next Split, Extend or Domain Split taken from a FIFO queue, unless its context unifier is redundant. A similar proof procedure has been described for the $\mathcal{ME}$ calculus in [5].

The following is our main result (see [6] for a more precise statement and proof).

Theorem 13 (Completeness). *For every fair derivation of Φ and Γ, the (limit) context of every exhausted branch of its limit tree induces a LIA-model of $\Phi \cup \Gamma$.*

Note that this result includes a proof convergence result, that *every* fair derivation of an unsatisfiable clause set is a refutation. In practical terms, it implies that as long as a derivation strategy guarantees fairness, the order of application of the rules of the calculus is irrelevant for proving an input clause set unsatisfiable, giving to the $\mathcal{ME}$(LIA) calculus the same flexibility enjoyed by the DPLL calculus at the propositional level.

An interesting special case arises when the exhausted branch in Theorem 13 is finite. The branch then readily provides a model of the input clause set.

6 Conclusions and Further Work

We have presented a basic version of $\mathcal{ME}(\mathrm{LIA})$, a new calculus for a logic with restricted quantifiers and linear integer constraints. The calculus allows one to reason with certain useful extensions of linear integer arithmetic with relations and finite domain constants. With the restriction of variables to finite domains, implementations of the calculus have potential applications in formal methods and in planning, where they can scale better than current decision procedures based on weaker logics, such as propositional logic or function-free clause logic.

We are working on extending the set of derivation rules with rules analogous to the unit-propagation rule of DPLL, which are crucial for producing efficient implementations. With that goal, we are also working on refinements of the calculus that reduce the cost of processing LIA-constraints. We stress though that the basic version presented here is already geared toward efficiency for featuring a (semantically justified) redundancy criterion, by reduction to LIA's ordering constraints, that allows one to avoid inferences with clause instances satisfied by one of the current candidate models.

References

1. Bachmair, L., Ganzinger, H., Waldmann, U.: Refutational Theorem Proving for Hierachic First-Order Theories. Appl. Algebra Eng. Commun. Comput. 5, 193–212 (1994)
2. Baumgartner, P.: Theory Reasoning in Connection Calculi. LNCS (LNAI), vol. 1527. Springer, Heidelberg (1998)
3. Baumgartner, P.: Logical Engineering with Instance-Based Methods. In: Pfenning, F. (ed.) CADE 2007. LNCS, vol. 4603, pp. 404–409. Springer, Heidelberg (2007)
4. Baumgartner, P., Fuchs, A., de Nivelle, H., Tinelli, C.: Computing Finite Models by Reduction to Function-Free Clause Logic. Journal of Applied Logic (in Press, 2007)
5. Baumgartner, P., Fuchs, A., Tinelli, C.: Implementing the Model Evolution Calculus. International Journal of Artificial Intelligence Tools 15(1), 21–52 (2006)
6. Baumgartner, P., Fuchs, A., Tinelli, C.: $\mathcal{ME}(\mathrm{LIA})$– Model Evolution With Linear Integer Arithmetic Constraints. Technical Report, Department of Computer Science, The University of Iowa (2008), http://www.cs.uiowa.edu/~tinelli
7. Baumgartner, P., Tinelli, C.: The Model Evolution Calculus. In: Baader, F. (ed.) CADE 2003. LNCS (LNAI), vol. 2741, pp. 350–364. Springer, Heidelberg (2003)
8. Bürckert, H.J.: A Resolution Principle for Clauses with Constraints. In: Stickel, M.E. (ed.) CADE 1990. LNCS (LNAI), vol. 449, pp. 178–192. Springer, Heidelberg (1990)
9. Ganzinger, H., Korovin, K.: Theory Instantiation. In: Hermann, M., Voronkov, A. (eds.) LPAR 2006. LNCS (LNAI), vol. 4246, pp. 497–511. Springer, Heidelberg (2006)
10. Ge, Y., Barrett, C., Tinelli, C.: Solving Quantified Verification Conditions Using Satisfiability Modulo Theories. In: Pfenning, F. (ed.) CADE 2007. LNCS, vol. 4603. Springer, Heidelberg (2007)

11. Korovin, K., Voronkov, A.: Integrating Linear Arithmetic Into Superposition Calculus. In: Duparc, J., Henzinger, T.A. (eds.) CSL 2007. LNCS, vol. 4646, pp. 223–237. Springer, Heidelberg (2007)
12. Antonio, J., Peréz, N.: Encoding and Solving Problems in Effectively Propositional Logic. PhD thesis, The University of Manchester (2007)
13. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT Modulo Theories: from an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). J. of the ACM 53(6), 937–977 (2006)
14. Nieuwenhuis, R., Rubio, A.: Paramodulation-Based Theorem Proving. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning. Elsevier/MIT press (2001)
15. Stickel, M.E.: Automated Deduction by Theory Resolution. J. of Aut. R. 1, 333–355 (1985)

A Constraint Sequent Calculus for First-Order Logic with Linear Integer Arithmetic

Philipp Rümmer

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden
philipp@chalmers.se

Abstract. First-order logic modulo the theory of integer arithmetic is the basis for reasoning in many areas, including deductive software verification and software model checking. While satisfiability checking for ground formulae in this logic is well understood, it is still an open question how the general case of quantified formulae can be handled in an efficient and systematic way. As a possible answer, we introduce a sequent calculus that combines ideas from free-variable constraint tableaux with the Omega quantifier elimination procedure. The calculus is complete for theorems of first-order logic (without functions, but with arbitrary uninterpreted predicates), can decide Presburger arithmetic, and is complete for a substantial fragment of the combination of both.

1 Introduction

One of the main challenges in automated theorem proving is to combine reasoning about full first-order logic (FOL), including quantifiers, with reasoning about theories like the integers. At the time, there are efficient provers for handling formulae in first-order logic, as well as SMT-solvers that can efficiently handle ground problems modulo many theories, but the support for the combination of both is typically weak. In this paper, we develop a novel calculus for reasoning about first-order logic modulo linear integer arithmetic that is complete for both the first-order part and the theory part, and that can handle a substantial fragment of the combination of both. Because the calculus is close to the DPLL(T) architecture, techniques and optimisations used in SMT-solvers are readily applicable when working on ground problems, but can be combined with free-variable techniques to treat quantifiers more systematically.

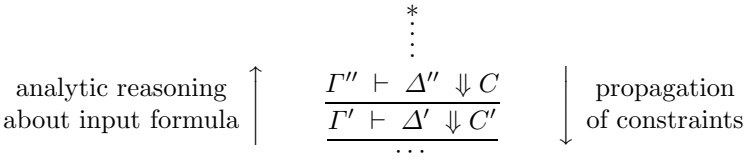
We start from two existing approaches: free-variable tableaux with incremental closure, following the work by Martin Giese [1], and the Omega quantifier elimination procedure [2] for deciding Presburger arithmetic (PA) [3]. From the former method, our calculus inherits the concept of generating *constraints* that describe valuations of free variables for which a formula is satisfied. The latter method provides the basic rules for dealing with linear integer arithmetic, and the concept of recursive application of a calculus in order to handle nested

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 274–289, 2008.

and alternating quantifiers. The resulting calculus accepts arbitrary formulae of PA enriched with arbitrary uninterpreted predicates as input. Uninterpreted functions are not directly supported, but can be treated by a translation to uninterpreted predicates and functionality and totality axioms.

Our calculus operates on *constrained sequents* $\Gamma \vdash \Delta \Downarrow C$, which consist of two sets Γ, Δ of formulae (the antecedent and the succedent) and one further formula C (the constraint). In this paper, C will always be a formula of PA. The semantics of a constrained sequent is the same as of the implication $C \Rightarrow (\Gamma \vdash \Delta)$, i.e., we call the sequent valid if the constraint C implies the ordinary sequent $\Gamma \vdash \Delta$ (and the ordinary sequent holds iff the formula $\bigwedge \Gamma \rightarrow \bigvee \Delta$ holds). In this sense, we can say that the constraint C is an approximation of the sequent $\Gamma \vdash \Delta$. The sequent $\forall x.(x \dot{\geq} 0 \rightarrow p(x)) \vdash p(c) \Downarrow c \dot{\geq} 0$ is valid, for instance, as are the sequents $\forall x.(x \dot{\geq} 0 \rightarrow p(x)) \vdash p(c) \Downarrow c \doteq 3$ and $\Gamma \vdash \Delta \Downarrow \mathit{false}$.

In practice, the constraints of sequents will be unknown during the construction of a proof. Proving thus consists of two or more phases: starting with a problem $\Gamma \vdash \Delta \Downarrow ?$ with unknown constraint, a proof procedure will first apply analytic rules to the antecedent and succedent and build a proof tree, similarly as in a normal Gentzen-style sequent calculus. At some point when it seems appropriate, the procedure will start to close branches by synthesising constraints, which are subsequently propagated downwards from the leaves to the root of the tree. If the constraint that reaches the root is found to be valid, the validity of the input problem $\Gamma \vdash \Delta$ has been shown; otherwise, the procedure will continue to expand the proof tree and later update the resulting constraints.



If the input problem $\Gamma \vdash \Delta$ does not contain uninterpreted predicates (i.e., corresponds to a PA formula), it is always possible to find proofs such that the resulting constraint is equivalent to $\Gamma \vdash \Delta$ (we will call such proofs *exhaustive*). This allows us to use the calculus as a quantifier elimination procedure for PA.

Our main contributions are: the introduction of the calculus, completeness results for a number of fragments (including FOL and PA), a complete and terminating proof strategy for the PA fragment, and the result that fair proof construction is complete for formulae that are provable at all. Proofs for all theorems in the paper are given in [4].

The paper is organised as follows: After giving basic definitions in Sect. 2, we introduce our calculus in three steps: Sect. 3 gives a version for pure first-order logic, Sect. 4 a minimalist version for first-order logic modulo integer arithmetic, together with completeness results, and Sect. 5 an equivalent but more refined calculus. Sect. 6 contains the result that fair proof strategies are complete. Finally, Sect. 7 summarises related work and Sect. 8 concludes.

2 Preliminaries

We assume that the reader is familiar with classical first-order logic and Gentzen-style sequent calculi, see [5] for an introduction. Assuming that $x \in X$ ranges over an infinite set of variables, $c \in A$ over an infinite set of constants, $p \in P$ over a set of uninterpreted predicates with fixed arity, and $\alpha \in \mathbb{Z}$ over integers, the syntactic categories of terms t and formulae ϕ are defined by:

$$t \;::=\; \alpha \mid x \mid c \mid \alpha t + \cdots + \alpha t$$
$$\phi \;::=\; \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \mid \forall x.\phi \mid \exists x.\phi \mid t \doteq 0 \mid t \dot{\geq} 0 \mid t \dot{\leq} 0 \mid \alpha \mid t \mid p(t, \ldots, t)$$

For reasons of simplicity, we only allow 0 as right-hand side of equations and inequalities, although we deviate from this convention in some places for sake of clarity. The explicit divisibility operator $\alpha \mid t$ is added for presentation purposes only and does not add any expressiveness (divisibility can also be expressed with an existentially quantified equation). Further, we use the abbreviations *true*, *false* for the equations $0 \doteq 0$, $1 \doteq 0$ and $\phi \rightarrow \psi$ as abbreviation for $\neg\phi \vee \psi$.

Simultaneous substitution of terms $t_1, \ldots, t_n$ for variables $x_1, \ldots, x_n$ is denoted by $[x_1/t_1, \ldots, x_n/t_n]\phi$, whereby we assume that variable capture is avoided by renaming bound variables when necessary. As short-hand notations, we sometimes also substitute terms for constants (as in $[c/t]\phi$), quantify over constants (as in $\forall c.\phi$), or quantify over sets of constants (as in $\forall U.\phi$).

Semantics. The only universe considered for evaluation are the integers $\mathbb{Z}$ (an exception is Sect. 3, where we treat normal first-order logic). A variable assignment $\beta : X \rightarrow \mathbb{Z}$ is a mapping from variables to integers, a constant assignment $\delta : A \rightarrow \mathbb{Z}$ a mapping from constants to integers, and an interpretation $I : P \rightarrow \mathcal{P}(\mathbb{Z}^*)$ a mapping from predicates to sets of $\mathbb{Z}$-tuples. The evaluation function $val_{I,\beta,\delta}$ for terms and formulae is then defined as is common and gives the arithmetic operations their normal meaning. We call a formula ϕ valid if $val_{I,\beta,\delta}(\phi)$ is true for all I, β, δ.

Sequents. If Γ, Δ are finite sets of formulae and C is a formula, all of which do not contain free variables, then $\Gamma \vdash \Delta$ is an (ordinary) sequent and $\Gamma \vdash \Delta \Downarrow C$ is a (constrained) sequent. We sometimes identify sequents with the formulae $\bigwedge \Gamma \rightarrow \bigvee \Delta$ (resp., $\bigwedge \Gamma \wedge C \rightarrow \bigvee \Delta$). A calculus rule is a binary relation between finite sets of constrained sequents (the premisses) and constrained sequents (the conclusion). A sequent calculus rule is called sound, iff, for all instances

$$\frac{\Gamma_1 \vdash \Delta_1 \Downarrow C_1 \quad \cdots \quad \Gamma_n \vdash \Delta_n \Downarrow C_n}{\Gamma \vdash \Delta \Downarrow C}$$

it holds that: if all premisses $\Gamma_1 \vdash \Delta_1 \Downarrow C_1$, …, $\Gamma_n \vdash \Delta_n \Downarrow C_n$ are valid, then $\Gamma \vdash \Delta \Downarrow C$ is valid. Proof trees are defined as is common as trees growing upwards in which each node is labelled with a constrained sequent, and in which each node that is not a leaf is related with the nodes directly above through an instance of a calculus rule. A proof is closed if it is finite, and if all leaves are justified by a rule instance without premisses.

Simplification. We denote elementary simplification steps on terms and atomic formulae in a proof with SIMP, without showing more details about the applied transformation (in an implementation, SIMP might be a part of the datastructures for formulae). SIMP normalises terms to the form $\alpha_1 t_1 + \cdots + \alpha_n t_n$, in which $\alpha_1, \ldots, \alpha_n$ are non-zero integers and $t_1, \ldots, t_n$ are pairwise distinct variables, constants, or 1 (possibly 0 as the empty sum). Further, terms are put into a canonical form by sorting summands according to a well-founded ordering $<_r$:

- on variables, constants and integers, $<_r$ is an arbitrary well-ordering such that variables are bigger than constants, constants are bigger than integers, and: $0 <_r 1 <_r -1 <_r 2 <_r -2 <_r 3 <_r \cdots$.
- on terms with coefficients, $<_r$ is defined by $\alpha t <_r \alpha' t'$ if and only if $t <_r t'$ or $t = t'$ and $\alpha <_r \alpha'$.
- on linear combinations, $<_r$ is defined by $\alpha_1 t_1 + \cdots + \alpha_n t_n <_r \alpha'_1 t'_1 + \cdots + \alpha'_k t'_k$ if and only if $\{\!\{\alpha_1 t_1, \ldots, \alpha_n t_n\}\!\} <_r \{\!\{\alpha'_1 t'_1, \ldots, \alpha'_n t'_n\}\!\}$ (in the multiset extension of $<_r$, cf. [6]).

Atomic formulae $t \doteq 0$, $t \dot{\geq} 0$, $t \dot{\leq} 0$ are normalised by SIMP such that the coefficients of non-constant terms in t are coprime (do not have non-trivial factors in common), and such that the leading coefficient is non-negative. This also detects that equations like $2y - 6c + 1 \doteq 0$ are unsolvable and equivalent to *false*, and that an inequality like $2y - 6c + 1 \dot{\leq} 0$ can be simplified and rounded to $y - 3c + 1 \dot{\leq} 0$ thanks to the discreteness of the integers. All inequalities in the succedent are moved to the antecedent. A divisibility judgement $\alpha \mid t$ is normalised like an equation $\alpha x + t \doteq 0$, and it is ensured that α and the leading coefficient of t are positive.

3 A Constraint Sequent Calculus for First-Order Logic

We first introduce a very restricted calculus for pure first-order logic, in order to illustrate how the framework of constrained sequents is related to normal free-variable tableau calculi. This section is exceptional in that we do *not* assume evaluation of formulae over the universe $\mathbb{Z}$ of integers, and that we allow equations $s \doteq t$ whose right-hand side is not 0. The rules from Fig. 1, together with the following closure rule, form the calculus Pred^C:

$$\frac{*}{\Gamma, p(s_1, \ldots, s_n) \vdash p(t_1, \ldots, t_n), \Delta \Downarrow \bigwedge_i s_i \doteq t_i} \text{ PRED-CLOSE}$$

Instead of unifying complementary literals, a conjunction of equations about the predicate arguments is generated and propagated as a constraint.

Example 1. We show a proof for the sequent $\forall x.\exists y.p(x, y) \vdash \exists z.p(a, z)$. In order to instantiate existential and universal quantifiers, fresh constants c, d, e are introduced. The constraints on the right-hand side are practically filled in

$$\frac{\Gamma \vdash \phi, \Delta \Downarrow C \qquad \Gamma \vdash \psi, \Delta \Downarrow D}{\Gamma \vdash \phi \wedge \psi, \Delta \Downarrow C \wedge D}\ \text{AND-RIGHT}$$

$$\frac{\Gamma, \phi \vdash \Delta \Downarrow C \qquad \Gamma, \psi \vdash \Delta \Downarrow D}{\Gamma, \phi \vee \psi \vdash \Delta \Downarrow C \wedge D}\ \text{OR-LEFT}$$

$$\frac{\Gamma, \phi, \psi \vdash \Delta \Downarrow C}{\Gamma, \phi \wedge \psi \vdash \Delta \Downarrow C}\ \text{AND-LEFT} \qquad \frac{\Gamma \vdash \phi, \psi, \Delta \Downarrow C}{\Gamma \vdash \phi \vee \psi, \Delta \Downarrow C}\ \text{OR-RIGHT}$$

$$\frac{\Gamma \vdash \phi, \Delta \Downarrow C}{\Gamma, \neg\phi \vdash \Delta \Downarrow C}\ \text{NOT-LEFT} \qquad \frac{\Gamma, \phi \vdash \Delta \Downarrow C}{\Gamma \vdash \neg\phi, \Delta \Downarrow C}\ \text{NOT-RIGHT}$$

$$\frac{\Gamma \vdash [x/c]\phi, \exists x.\phi, \Delta \Downarrow [x/c]C}{\Gamma \vdash \exists x.\phi, \Delta \Downarrow \exists x.C}\ \text{EX-RIGHT} \qquad \frac{\Gamma, [x/c]\phi, \forall x.\phi \vdash \Delta \Downarrow [x/c]C}{\Gamma, \forall x.\phi \vdash \Delta \Downarrow \exists x.C}\ \text{ALL-LEFT}$$

$$\frac{\Gamma \vdash [x/c]\phi, \Delta \Downarrow [x/c]C}{\Gamma \vdash \forall x.\phi, \Delta \Downarrow \forall x.C}\ \text{ALL-RIGHT} \qquad \frac{\Gamma, [x/c]\phi \vdash \Delta \Downarrow [x/c]C}{\Gamma, \exists x.\phi \vdash \Delta \Downarrow \forall x.C}\ \text{EX-LEFT}$$

Fig. 1. The rules for first-order predicate logic (without equality). In all rules, c is a constant that does not occur in the conclusion: in contrast to the usage of Skolem functions and free variables in tableaux, the same kinds of symbols (constants) are used to handle both existential and universal quantifiers. Arbitrary renaming of bound variables is allowed in the constraints when necessary to avoid variable capture.

after applying PRED-CLOSE. Because $\exists x.\forall y.\exists z.(x \doteq a \wedge y \doteq z)$ is valid, also the validity of the original problem is proven.

$$\cfrac{\cfrac{\cfrac{\cfrac{*}{\ldots, p(c,d) \vdash \ldots, p(a,e) \Downarrow c \doteq a \wedge d \doteq e}\ \text{PRED-CLOSE}}{\ldots, p(c,d) \vdash \exists z.p(a,z) \Downarrow \exists z.(c \doteq a \wedge d \doteq z)}\ \text{EX-RIGHT}}{\ldots, \exists y.p(c,y) \vdash \exists z.p(a,z) \Downarrow \forall y.\exists z.(c \doteq a \wedge y \doteq z)}\ \text{EX-LEFT}}{\forall x.\exists y.p(x,y) \vdash \exists z.p(a,z) \Downarrow \exists x.\forall y.\exists z.(x \doteq a \wedge y \doteq z)}\ \text{ALL-LEFT}$$

It is easy to see that a constraint C produced by a proof can only consist of equations over variables and constants, conjunctions, and quantifiers (because these are the only constructs that are introduced in constraints by the rules of Pred^C). The validity of constraints/formulae of this kind is decidable and corresponds to simultaneous unification, which makes the calculus effective.

Lemma 2 (Soundness). *If a sequent $\Gamma \vdash \Delta \Downarrow C$ is provable in Pred^C, then it is valid (holds in all first-order structures).*

Lemma 3 (Completeness). *Suppose ϕ is closed, valid (holds in all first-order structures), and does not contain constants. Then there is a valid constraint C such that $\vdash \phi \Downarrow C$ is provable in Pred^C.*

4 Adding Integer Arithmetic

Relatively few changes to the calculus Pred^C from the previous section are necessary to reason about problems in integer arithmetic. In this section, we describe a minimalist approach in which all integer reasoning happens during the constraint solving and investigate fragments on which the resulting method is complete. Later in the paper, the calculus is refined and optimised. From now on and in contrast to the previous section, assume that formulae and terms are evaluated over first-order structures with the universe $\mathbb{Z}$ as described in Sect. 2.

In contrast to the previous section, to handle integer arithmetic disjunctive constraints also need to be considered. We thus split the rule PRED-CLOSE into two new rules, one of which (PRED-UNIFY) generates unification conditions for complementary pairs, while the other one (CLOSE) allows to synthesise a constraint from arbitrary formulae in a sequent:

$$\frac{\Gamma, p(s_1,\ldots,s_n) \vdash p(t_1,\ldots,t_n), \bigwedge_i s_i - t_i \doteq 0, \Delta \Downarrow C}{\Gamma, p(s_1,\ldots,s_n) \vdash p(t_1,\ldots,t_n), \Delta \Downarrow C}\ \text{PRED-UNIFY}$$

$$\frac{*}{\Gamma, \phi_1,\ldots,\phi_n \vdash \psi_1,\ldots,\psi_m, \Delta \Downarrow \neg\phi_1 \vee \cdots \vee \neg\phi_n \vee \psi_1 \vee \cdots \vee \psi_m}\ \text{CLOSE}$$

$(\phi_1,\ldots,\phi_n,\psi_1,\ldots,\psi_m$ do not contain uninterpreted predicates)

Besides these two rules, $\mathrm{PresPred}^C_S$ contains all rules given in Fig. 1. It is obvious that any proof in Pred^C can be translated to a proof in $\mathrm{PresPred}^C_S$ by replacing applications of PRED-CLOSE with applications of PRED-UNIFY, followed by CLOSE, which means that $\mathrm{PresPred}^C_S$ is complete for first-order logic.

Because uninterpreted predicates are excluded in CLOSE, the constraint resulting from a proof is always a formula in Presburger arithmetic and can in principle be handled using any decision procedure for PA (e.g. [2], also see Sect. 5.3). We come back to this issue later in the paper and assume for the time being that some procedure is available for deciding the validity of constraints.

As an implication of a more general result (Lem. 13), it can be observed that $\mathrm{PresPred}^C_S$ is proof-confluent: if ϕ is provable, then every partial proof of $\vdash \phi \Downarrow ?$ can be extended to a closed proof of a sequent $\vdash \phi \Downarrow C$ with valid constraint C.

Example 4. We show a proof for the following sequent (Fig. 2):

$$\forall x.p(2x), \forall x.\neg p(2x+1) \vdash \forall y.(p(y) \rightarrow p(y+10))$$

The sequent is proven by first building the "main proof" (upwards) to a point where CLOSE can be applied. The constraints $C_1, \ldots, C_4$ are then filled in and propagated downwards. Because C_4 is valid, we have proven the validity of the original formula. The constraint simplification is explained in more detail later.

Completeness on fragments. Two fragments on which $\mathrm{PresPred}^C_S$ is complete are the classes of purely universal and of purely existential formulae. We call positions in the antecedent/succedent of a sequent *positive* if they are underneath an odd/even number of negations. All other positions are called *negative*.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{*}{\ldots \vdash \ldots, 2d-c-10 \doteq 0, c-2e-1 \doteq 0 \Downarrow C_1}\,\text{CLOSE}}{p(2d),\ldots,p(c) \vdash p(c+10), p(2e+1) \Downarrow C_1}\,\text{PRED-UNIFY} \times 2}{\ldots, p(2d), \forall x.\neg p(2x+1), p(c) \vdash p(c+10) \Downarrow C_2}\,\text{ALL-LEFT, NOT-LEFT}}{\forall x.p(2x), \forall x.\neg p(2x+1), p(c) \vdash p(c+10) \Downarrow C_3}\,\text{ALL-LEFT}}{\forall x.p(2x), \forall x.\neg p(2x+1) \vdash \neg p(c) \vee p(c+10) \Downarrow C_3}\,\text{OR-RIGHT, NOT-RIGHT}}{\forall x.p(2x), \forall x.\neg p(2x+1) \vdash \forall y.(p(y) \rightarrow p(y+10)) \Downarrow C_4}\,\text{ALL-RIGHT}$$

The constraints are:

$$\begin{array}{lcll} C_1 & = & & 2d-c-10 \doteq 0 \vee c-2e-1 \doteq 0 \\ C_2 & = & \exists y.[e/y]C_1 & = & \exists y.(2d-c-10 \doteq 0 \vee c-2y-1 \doteq 0) \\ C_3 & = & \exists x.[d/x]C_2 & = & \exists x.\exists y.(2x-c-10 \doteq 0 \vee c-2y-1 \doteq 0) \\ & & & \equiv & 2 \mid (c+10) \vee 2 \mid (c-1) \\ C_4 & = & \forall x.[c/x]C_3 & = & \forall x.(2 \mid (x+10) \vee 2 \mid (x-1)) \\ & & & \equiv & \mathit{true} \end{array}$$

Fig. 2. An example proof in the calculus PresPred_S^C

Lemma 5. *If $\Gamma \vdash \Delta$ is a valid sequent in which $\exists$ only occurs in negative and $\forall$ only in positive positions, then there is a valid PA constraint C such that $\Gamma \vdash \Delta \Downarrow C$ has a proof in the calculus PresPred_S^C.*

Lemma 6. *If $\Gamma \vdash \Delta$ is a valid sequent (without constants) in which $\exists$ only occurs in positive and $\forall$ only in negative positions, then there is a valid PA constraint C such that $\Gamma \vdash \Delta \Downarrow C$ has a proof in the calculus PresPred_S^C.*

Comparison with $\mathcal{ME}$(LIA). We can also show that the calculus PresPred_S^C is complete on the logic that can be handled by Model Evolution modulo linear integer arithmetic [7]. Ignoring minor syntactic issues and the fact that $\mathcal{ME}$(LIA) works on clauses, $\mathcal{ME}$(LIA) is a sound and complete calculus for proving the unsatisfiability of formulae of the shape $\exists \bar{a}.(\phi \wedge \psi)$, where $\bar{a} = (a_1, \ldots, a_m)$ is a vector of existentially quantified variables, ϕ is a PA formula over $\bar{a}$ that only has finitely many solutions, and ψ is an arbitrary formula over $\bar{a}$ in which $\exists/\forall$ only occurs in negative/positive positions.

Lemma 7. *If $\exists \bar{a}.(\phi \wedge \psi)$ as above is an unsatisfiable formula that does not contain constants or free variables, then there is a valid constraint C such that the sequent $\exists \bar{a}.(\phi \wedge \psi) \vdash \Downarrow C$ has a proof in PresPred_S^C.*

5 Built-In Handling of Presburger Arithmetic

Although the calculus from the previous section is in principle usable, it practically has a number of shortcomings: the handling of arithmetic in constraints provides little guidance for the construction of proofs, so that large constraints are produced in a very indeterministic manner that cannot be solved efficiently.

Moreover, constraints are even needed to handle ground problems, for which branch-local reasoning should be sufficient. The main goal when refining the calculus is, therefore, to reduce the usage of constraints as far as possible.

In this section, we define built-in rules for handling linear integer arithmetic that can be interleaved with the rules from the previous section. The rules make it possible to handle ground problems branch-locally: proof trees for ground problems can be constructed depth-first (non-iteratively), similarly to the way in which SMT-solvers work. It can be achieved that the only constraints that can result from a subproof in case of ground problems are *true* or *false* (more details are given in [4]). Branch-local reasoning is also possible for innermost $\forall$-quantifiers in positive and $\exists$ in negative positions. The arithmetic rules also yield a decision procedure for PA that can be used to decide constraints (Sect. 5.3).

The rules in detail. The calculus $\mathrm{PresPred}^C$ consists of the rules given in Fig. 3, together with all rules from the calculus $\mathrm{PresPred}^C_S$ and the simplification rule SIMP. We introduce new rules EX-RIGHT-D, ALL-LEFT-D that instantiate quantified formulae destructively, because formulae that do not contain uninterpreted predicates never have to be instantiated twice (also see Lem. 13 below).

The equality handling follows the calculus given in [8] and can solve arbitrary equations in the antecedent, in the sense that the equations are rewritten until the leading coefficients are all 1 and the leading terms of equations occur in exactly one place. Speaking in terms of matrices, RED is the rule for performing row operations, while COL-RED(-SUBST) is responsible for column operations. We define a suitable strategy for guiding the rules below.

The rules DIV-RIGHT and DIV-LEFT translate divisibility statements to equations, while DIV-CLOSE synthesises divisibility statements from equations. The formula C' in DIV-CLOSE can be found through pseudo-division (multiplying equations, inequalities or divisibility statements in C with non-zero factors). For $C = (c + d \doteq 0)$ and $\alpha = 3$, for instance, we would choose $C' = (x + 3d \doteq 0)$.

Inequalities are handled based on the Omega test [2], which is an extension of the Fourier-Motzkin variable elimination method (cf. [9]) for integer problems. The central rule is OMEGA-ELIM for replacing a conjunction of inequalities with a disjunction over simpler cases (OMEGA-ELIM is directly based on the main theorem underlying the Omega test [2]). The literal m_i in the rule is defined by:

$$m = \max_j \beta_j, \qquad m_i = \left\lfloor \frac{m\alpha_i - \alpha_i - m}{m} \right\rfloor$$

In case there are no upper bounds, we define $m = m_i = -1$. The application of OMEGA-ELIM is only meaningful if c does not occur in formulae other than inequalities. Note, that if there are no lower or no upper bounds, the rule will replace all inequalities whose leading term is c with *true*.

Because we avoid the application of OMEGA-ELIM in certain common situations (for instance, whenever the constant c occurs as argument of uninterpreted predicates), we also introduce a rule FM-ELIM for normal Fourier-Motzkin elimination. FM-ELIM can be applied with higher priority than OMEGA-ELIM and is often able to close proofs faster than OMEGA-ELIM, reducing the need to resort

$$\frac{\Gamma \vdash [x/c]\phi, \Delta \Downarrow [x/c]C}{\Gamma \vdash \exists x.\phi, \Delta \Downarrow \exists x.C}\ \text{EX-RIGHT-D} \qquad \frac{\Gamma, [x/c]\phi \vdash \Delta \Downarrow [x/c]C}{\Gamma, \forall x.\phi \vdash \Delta \Downarrow \exists x.C}\ \text{ALL-LEFT-D}$$

(c a constant that does not occur in the conclusion, ϕ does not contain uninterpreted predicates)

$$\frac{\Gamma, t \doteq 0 \vdash \phi[s + \alpha \cdot t], \Delta \Downarrow C}{\Gamma, t \doteq 0 \vdash \phi[s], \Delta \Downarrow C}\ \text{RED}$$

(α a literal, or t a literal and α an arbitrary term)

$$\frac{\Gamma, \alpha(u + c') + t \doteq 0, c - u - c' \doteq 0 \vdash \Delta \Downarrow [x/c']C}{\Gamma, \alpha c + t \doteq 0 \vdash \Delta \Downarrow \forall x.C}\ \text{COL-RED}$$

(c' a constant that does not occur in the conclusion or in u)

$$\frac{\Gamma, \alpha(u + c') + t \doteq 0, c - u - c' \doteq 0 \vdash \Delta \Downarrow [x/c']C}{\Gamma, \alpha c + t \doteq 0 \vdash \Delta \Downarrow [x/c - u]C}\ \text{COL-RED-SUBST}$$

(c' a constant that does not occur in the conclusion or in u)

$$\frac{\Gamma, \exists x.\alpha x + t \doteq 0 \vdash \Delta \Downarrow C}{\Gamma, \alpha \mid t \vdash \Delta \Downarrow C}\ \text{DIV-LEFT}$$

(x an arbitrary variable)

$$\frac{\Gamma, (\alpha \mid t + 1) \vee \cdots \vee (\alpha \mid t + \alpha - 1) \vdash \Delta \Downarrow C}{\Gamma \vdash \alpha \mid t, \Delta \Downarrow C}\ \text{DIV-RIGHT} \qquad (\alpha > 0)$$

$$\frac{\Gamma, \alpha c - t \doteq 0 \vdash \Delta \Downarrow C}{\Gamma, \alpha c - t \doteq 0 \vdash \Delta \Downarrow [x/t]C' \vee \alpha \nmid t}\ \text{DIV-CLOSE}$$

(c does not occur in t or in C', C' a PA formula such that $C \Leftrightarrow [x/\alpha c]C'$)

$$\frac{\Gamma \vdash t \dot{\leq} 0, \Delta \Downarrow C \quad \Gamma \vdash t \dot{\geq} 0, \Delta \Downarrow D}{\Gamma \vdash t \doteq 0, \Delta \Downarrow C \wedge D}\ \text{SPLIT-EQ}$$

$$\frac{\Gamma, t \doteq 0 \vdash \Delta \Downarrow C}{\Gamma, t \dot{\leq} 0, t \dot{\geq} 0 \vdash \Delta \Downarrow C}\ \text{ANTI-SYMM}$$

$$\frac{\Gamma, \alpha c + s \dot{\geq} 0, \beta c + t \dot{\leq} 0, \beta s - \alpha t \dot{\geq} 0 \vdash \Delta \Downarrow C}{\Gamma, \alpha c + s \dot{\geq} 0, \beta c + t \dot{\leq} 0 \vdash \Delta \Downarrow C}\ \text{FM-ELIM}$$

$(\alpha > 0,\ \beta > 0)$

$$\frac{\Gamma, \begin{array}{c} \bigwedge_{i,j} \alpha_i b_j - a_i \beta_j - (\alpha_i - 1)(\beta_j - 1) \dot{\geq} 0 \\ \vee \\ \bigvee_i \bigvee_{k=0}^{m_i} \left(\begin{array}{c} \alpha_i c - a_i - k \doteq 0 \wedge \\ \bigwedge_i \alpha_i c - a_i \dot{\geq} 0 \wedge \bigwedge_j \beta_j c - b_j \dot{\leq} 0 \end{array} \right) \end{array} \vdash \Delta \Downarrow C}{\Gamma, \{\alpha_i c - a_i \dot{\geq} 0\}_i, \{\beta_j c - b_j \dot{\leq} 0\}_j \vdash \Delta \Downarrow C}\ \text{OMEGA-ELIM}$$

$(\alpha_i > 0,\ \beta_j > 0)$

Fig. 3. Rules for equations, inequalities, and divisibility judgements. In RED, we write $\phi[s]$ in the succedent to denote that s occurs in an arbitrary formula in the sequent, which can in particular also be in the antecedent. m_i in OMEGA-ELIM as on page 281.

to the more complex rule. Further, we define two rules to convert between equations and inequalities. While the rule SPLIT-EQ is strictly necessary for certain problems, ANTI-SYMM is introduced only for reasons of efficiency.

Lemma 8 (Soundness). *If a sequent* $\Gamma \vdash \Delta \Downarrow C$ *is provable in* $\mathit{PresPred}^C$*, then it is valid.*

5.1 Exhaustive Proofs

The existence of a closed proof for a sequent $\Gamma \vdash \Delta \Downarrow C$ guarantees that the implication $C \Rightarrow (\Gamma \vdash \Delta)$ holds (this is the soundness of the calculus, Lem. 8). In the special case that the sequent $\Gamma \vdash \Delta$ does not contain uninterpreted predicates, it is possible to distinguish particular closed proofs that also guarantee the opposite implication $(\Gamma \vdash \Delta) \Rightarrow C$, and thus $(\Gamma \vdash \Delta) \Leftrightarrow C$. While this can be achieved in a trivial way by always applying CLOSE such that *all* formulae in a sequent are selected, it is sufficient to impose a weaker condition on proof trees that leads to smaller constraints and also makes it possible to eliminate quantifiers (Sect. 5.3). To this end, it is necessary to remember whether a constant was introduced by an existential rule (like EX-RIGHT) or a universal rule (like ALL-RIGHT). A generalisation of the condition is described in [4].

Assume that a $\mathrm{PresPred}^C$-proof is given. We annotate the sequents in the proof with sets U of "universal" constants that the calculus attempts to eliminate. More formally, the proof is called *exhaustive* iff there is a mapping from proof nodes (constrained sequents) to sets U of constants that satisfies:

1. The rules AND-*, OR-*, NOT-*, PRED-UNIFY, RED, DIV-*, SPLIT-EQ, ANTI-SYMM, FM-ELIM, and SIMP keep or reduce the set: if the conclusion is annotated with U, the premisses are annotated with arbitrary subsets of U.
2. The rules EX-RIGHT(-D), ALL-LEFT(-D) erase the set: the premiss is annotated with $\emptyset$.
3. The rules EX-LEFT and ALL-RIGHT may add the introduced constant c to the set: if the conclusion is annotated with U, then the premiss is annotated with a subset of $U \cup \{c\}$.
4. The rule COL-RED is only applied if the conclusion is annotated with U such that $c \in U$. In this case, the premiss is annotated with a subset of $U \cup \{c'\}$.
5. The rule COL-RED-SUBST is only applied if the conclusion is annotated with U such that $c \notin U$, and if u does not contain any constants from U. In this case, the premiss is annotated with a subset of U.
6. The rule OMEGA-ELIM is only applied if the conclusion is annotated with U such that $c \in U$ and if c does not occur in Γ or Δ. In this case, the premiss is annotated with an arbitrary subset of U.
7. The rule DIV-CLOSE is only applied if the conclusion is annotated with U such that $c \in U$. In this case, the premiss is annotated with a subset of U.
8. The rule CLOSE is always applied such that all formulae without uninterpreted predicates are selected, apart from (possibly) those equations in the succedent that contain constants from U that exclusively occur in equations in the succedent.

Lemma 9 (Constraint completeness). *Suppose that a PresPred^C-proof is closed and exhaustive. For each sequent $\Gamma \vdash \Delta \Downarrow C$ in the tree, let Γ_p, Δ_p denote the subsets of PA formulae in Γ, Δ. Then, for each sequent $\Gamma \vdash \Delta \Downarrow C$ that is annotated with a set U, the implication $\forall U.\ (\Gamma_p \vdash \Delta_p) \Rightarrow \forall U.\ C$ holds.*

Example 10. The formula $\neg\exists x.\exists y.(2x - c - 10 \doteq 0 \vee 2y - c + 1 \doteq 0)$ from Example 4 is simplified by constructing a proof. To see that the proof is exhaustive, the sequent with constraint D_5 is annotated with $\emptyset$, the sequent with D_1 with $\{e\}$, the sequent with D_3 with $\{d\}$, and all other sequents with the set $\{d, e\}$. This implies that the original formula is equivalent to D_5.

$$\dfrac{\dfrac{\dfrac{*}{2d - c - 10 \doteq 0 \vdash \ \Downarrow D_1}\ \text{CLOSE}}{2d - c - 10 \doteq 0 \vdash \ \Downarrow D_2}\ \text{DIV-CLOSE} \qquad \dfrac{\dfrac{*}{2e - c + 1 \doteq 0 \vdash \ \Downarrow D_3}\ \text{CLOSE}}{2e - c + 1 \doteq 0 \vdash \ \Downarrow D_4}\ \text{DIV-CLOSE}}{\dfrac{2d - c - 10 \doteq 0 \vee 2e - c + 1 \doteq 0 \vdash \ \Downarrow D_2 \wedge D_4}{\exists x.\exists y.(2x - c - 10 \doteq 0 \vee 2y - c + 1 \doteq 0) \vdash \ \Downarrow D_5}\ \text{EX-LEFT} \times 2}\ \text{OR-LEFT}$$

The constraints resulting from the proof are:

$$\begin{array}{lcll}
D_1 & = & & 2d - c - 10 \not\doteq 0 \\
D_2 & = & [2d/c + 10]D_1 \vee 2 \nmid (c + 10) & (c + 10) - c - 10 \not\doteq 0 \vee 2 \nmid (c + 10) \\
 & & & \equiv\ 2 \nmid (c + 10) \\
D_3 & = & & 2e - c + 1 \not\doteq 0 \\
D_4 & = & [2e/c - 1]D_3 \vee 2 \nmid (c - 1) & (c - 1) - c + 1 \not\doteq 0 \vee 2 \nmid (c - 1) \\
 & & & \equiv\ 2 \nmid (c - 1) \\
D_5 & = & \exists x.[d/x]\exists y.[e/y](D_2 \wedge D_4) & \exists x.\exists y.(2 \nmid (c + 10) \wedge 2 \nmid (c - 1)) \\
 & & & \equiv\ 2 \nmid (c + 10) \wedge 2 \nmid (c - 1)
\end{array}$$

5.2 The Construction of Exhaustive Proofs for PA Problems

We define a strategy to apply the PresPred^C-rules to a sequent $\Gamma \vdash \Delta \Downarrow ?$ that only contains PA formulae. The strategy is guaranteed to terminate and to produce a closed and exhaustive proof, and it is deterministic in the sense that no search is required, every ordering of rule applications (that is consistent with given priorities) leads to an exhaustive proof. In order to guide the proof construction, the strategy maintains a set U of constants (which is initially empty) and a term ordering $<_r$ (as in Sect. 2) that are updated when new constants are introduced or existing constants need to be reordered. The ordering $<_r$ is always chosen such that the constants in U are bigger than all constants that are not in U. Both U and $<_r$ are branch-local: different branches in a proof tree can be built using different Us and $<_r$s.

We list the rules that the strategy applies to a proof goal with descending priority: step 2 will only be carried out if step 1 is impossible, etc.

1. apply SIMP (if possible).
2. apply RED if an α exists such that $s + \alpha \cdot t <_r s$
   (and if $s \neq t$ or $\phi[s]$ is not an equation in the antecedent).

3. if the antecedent contains an equation $\alpha c + t \doteq 0$ with $\alpha > 1$, then:
   - if $c \notin U$, apply COL-RED-SUBST. The fresh constant c' is inserted in the term ordering $<_r$ such that it becomes minimal, and u is chosen such that $(\alpha u + t) = \min_{<_r} \{\alpha u' + t \mid u' \text{ a term}\}$.
   - if $c \in U$ and t contains at least one further constant from U whose coefficient is not a multiple of α, apply COL-RED. The fresh constant c' is added to U and is inserted in the term ordering $<_r$ such that it becomes smaller than all other constants in U, but bigger than all constants not in U. u is again chosen such that $(\alpha u + t) = \min_{<_r} \{\alpha u' + t \mid u' \text{ a term}\}$.
4. if the antecedent contains an equation $\alpha c + t \doteq 0$ with $c \in U$, apply DIV-CLOSE, remove c from U, and update $<_r$ such that c becomes minimal. (This is also possible for $\alpha = 1$)
5. if possible, apply any of the following rules:
   - ANTI-SYMM.
   - FM-ELIM, if the result is not subsumed by an inequality in the antecedent.
   - any of the rules AND-*, OR-*, NOT-*.
6. if possible, apply any of the following rules:
   - SPLIT-EQ: if an equation can be split that contains a constant $c \in U$ that also occurs as leading term of an inequality in the antecedent.
   - OMEGA-ELIM: if inequalities $\{\alpha_i c - a_i \dot{\geq} 0\}_i$, $\{\beta_j c - b_j \dot{\leq} 0\}_j$ occur in the antecedent and $c \in U$, and if c does not occur in any other formula.
   - ALL-RIGHT, EX-LEFT: add the fresh constant c to U and insert it into $<_r$ such that it becomes maximal.
   - EX-RIGHT-D, ALL-LEFT-D: set U to $\emptyset$ and insert c arbitrarily into $<_r$.
   - DIV-LEFT, DIV-RIGHT.
7. apply CLOSE and select exactly those formulae that do not contain constants from U or uninterpreted predicates.

The steps 1–4 of the strategy work by eliminating all U-constants that occur in equations in the antecedent. Similarly as in [8], in the antecedent only equations will be left whose leading coefficient is 1 and whose leading term does not occur in other places in the sequent anymore. The steps 5–6 handle inequalities by first applying the Fourier-Motzkin rule exhaustively, and by eliminating constants using the Omega rule whenever possible. Also quantifiers, propositional connectives and divisibility judgements are treated in step 5–6. A proof that is constructed using this procedure is shown in Example 10.

Lemma 11 (Termination and exhaustiveness). *If a sequent $\Gamma \vdash \Delta \Downarrow ?$ does not contain uninterpreted predicates, the strategy from above terminates and produces a closed exhaustive proof.*

5.3 Deciding Presburger Arithmetic by Recursive Proving

The anticipated way to decide constraints in proofs is to eliminate quantifiers already during the constraint propagation, i.e., at the points where the rules EX-RIGHT(-D), ALL-LEFT(-D), ALL-RIGHT, EX-LEFT or COL-RED are applied and

cause quantifiers to occur in constraints. By eliminating such quantifiers right away, each subproof of the proof can be annotated with a constraint that is a quantifier-free PA formula. When building proofs incrementally, this makes it possible to easily distinguish between unsatisfiable subproofs (i.e., subproofs with an unsatisfiable constraint) that need to be expanded further, and satisfiable subproofs whose expansion can be postponed. Besides, due to Lem. 11 and as only quantifier-free constraints occur, the resulting procedure decides PA.

To eliminate quantifiers, the calculus PresPred^C can be used (Example 10):

Lemma 12 (Quantifier elimination). *Suppose a formula ϕ does not contain uninterpreted predicates and $\forall$ occurs in ϕ only in positive positions and $\exists$ only in negative positions. The strategy from the previous section produces a proof with root $\vdash \phi \Downarrow C$ in which C does not contain quantifiers (more precisely, if C contains a quantified subformula $Qx.\psi$, then x does not occur in ψ).*

6 Fair Construction of Proofs

We now compare the calculus PresPred^C with the more restricted calculus PresPred^C_S from Sect. 4. Because the former calculus is a superset of the latter, it is a trivial observation that any sequent provable in PresPred^C_S is also provable in PresPred^C. It can also be shown that PresPred^C cannot prove more sequents than PresPred^C_S [4], which means that the two calculi are equivalent.

Proofs in PresPred^C can be found by a backtracking-free fair application strategy. To define the notion "fair," it has to be observed that formulae in a PresPred^C-proof can be rewritten by applying RED or SIMP. When this happens, it is possible to identify a unique successor of the modified formula in the premiss of the rule application (vice versa, a formula can have multiple predecessors because distinct formulae could become equal when applying a rule).

A *fair PresPred^C-proof* for a sequent $\Gamma \vdash \Delta \Downarrow ?$ is a possibly infinite proof in PresPred^C in which all constraints are ? and all branches have the properties:

- *Fair treatment of formulae with uninterpreted predicates:* whenever at some point on the branch one of the rules in Fig. 1 is applicable to a formula that contains uninterpreted predicates, the rule is applied to the formula or to a successor of the formula at some later point on the branch. (This implies that ALL-LEFT and EX-RIGHT are applied infinitely often to each universally quantified formula with uninterpreted predicates).
- *Fair unification of complementary literals:* if there is a sequent on the branch of the shape $\Gamma, p(\bar{t}) \vdash p(\bar{s}), \Delta \Downarrow ?$, the rule PRED-UNIFY is applied at least once on the branch to the pair $p(\bar{t})$, $p(\bar{s})$ or to successors of these formulae.
- *Exhaustiveness:* all proof nodes can be annotated with sets U as in Sect. 5.1.

A constraint C is *generated* by a fair proof of $\Gamma \vdash \Delta \Downarrow ?$ if a (finite) proof for $\Gamma \vdash \Delta \Downarrow C$ can be obtained by chopping off all branches of the fair proof at some point, applying CLOSE in some way to the leaves and propagating the resulting constraints through the proof. The next lemma, together with Lem. 9, implies the completeness of a fair rule, formula, and branch selection strategy.

Lemma 13 (Fair construction). *Suppose that a* $\mathrm{PresPred}_S^C$*-proof for the sequent* $\Gamma \vdash \Delta \Downarrow C$ *exists. Every fair* $\mathrm{PresPred}^C$*-proof of* $\Gamma \vdash \Delta \Downarrow ?$ *whose root is annotated with the set* U *generates a constraint* D *with* $\forall U.C \Rightarrow \forall U.D$.

7 Related Work

$\mathcal{ME}$(LIA) [7] is a recently proposed variant of the Model Evolution calculus that is similar to our calculus in that it supports PA enhanced with uninterpreted predicates (and without functions) as input language, and that its architecture resembles tableau calculi. Model Evolution does not use rigid free variables that are shared among different branches in the way tableaux do, however, which means that also constraints can be kept branch-local. Further differences are that $\mathcal{ME}$(LIA) works on clauses, only supports a restricted form of existential quantification, and has a more explicit representation of candidate models.

SMT-solvers based on the DPLL(T) architecture [10] can handle ground problems modulo integer arithmetic (and many other theories) efficiently, but only offer heuristic quantifier handling. Because of the similarity between DPLL and sequent calculi, the work presented in this paper can be seen as an alternative approach to handling quantifiers that should also be applicable to DPLL(T).

The simplification of formulae by the rules in Fig. 3 is roughly comparable with deduction modulo [11]. The concept is here integrated in a setting that resembles free-variable tableaux to treat quantifiers more efficiently.

An approach to embed algebraic constraints in tableau calculi is described in [12], where quantifier elimination tasks in real arithmetic (possibly involving more than one proof goal) are carried out by an external procedure, in a manner comparable to the simultaneous solving of constraints from multiple proof goals described here. Uninterpreted functions or predicates are not handled.

There are a number of approaches to include theories into resolution-based calculi. [13] works with constraints that are solved in a theory, but requires to enumerate the solutions of constraints (whereas it is enough to check the validity of constraints in our work). In [14], while it is enough to check satisfiability of constraints, no uninterpreted functions or predicates are supported. A recent calculus to handle rational arithmetic is given in [15], and is similar to our work in that it has built-in rules to solve systems of equations and inequalities (based on Fourier-Motzkin). The calculus is complete under restrictions that effectively prevent quantification over rationals. It remains to be investigated how this fragment is related to the fragments discussed here.

8 Conclusions and Future Work

We have presented a novel calculus to reason about problems in first-order logic modulo linear integer arithmetic. The calculus is complete for function-free first-order logic (on such problems, proofs in the calculus resemble free-variable tableaux with incremental closure [1]) and can decide Presburger arithmetic (in a manner that is similar to the Omega test [2]). As further results, we have shown

that the calculus is at least as complete as the calculus $\mathcal{ME}(\mathrm{LIA})$, and allows the fair construction of proofs. An implementation of the calculus is available under `http://www.cs.chalmers.se/~philipp/princess` and is described in [4].

Apart from continuing the implementation and benchmarks, there are a number of concepts that require more research, among others: the encoding and handling of functions and further theories; the integration of lemma learning; the integration of connectivity conditions to make proof search more directed; the elimination of cuts in proofs; an analysis of the complexity of the calculus as a decision procedure for PA. We also plan to extend our calculus to support nonlinear arithmetic (following the work in [8]), and possibly rational arithmetic.

Acknowledgements. I want to thank Wolfgang Ahrendt, Nikolaj Bjørner, Richard Bubel, Reiner Hähnle, Henrik Johansson, and the anonymous referees for discussions and/or comments during different stages of this work.

References

1. Giese, M.: Incremental closure of free variable tableaux. In: Goré, R.P., Leitsch, A., Nipkow, T. (eds.) IJCAR 2001. LNCS, vol. 2083, pp. 545–560. Springer, Heidelberg (2001)
2. Pugh, W.: The Omega test: a fast and practical integer programming algorithm for dependence analysis. In: Proceedings, 1991 ACM/IEEE conference on Supercomputing, pp. 4–13. ACM, New York (1991)
3. Presburger, M.: Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In: Sprawozdanie z I Kongresu metematyków slowiańskich, Warszawa, Warsaw, Poland, vol. 1929, pp. 92–101, 395 (1930)
4. Rümmer, P.: Calculi for Program Incorrectness and Arithmetic. PhD thesis, Chalmers University of Technology (to appear, 2008)
5. Fitting, M.C.: First-Order Logic and Automated Theorem Proving, 2nd edn. Springer, New York (1996)
6. Dershowitz, N., Manna, Z.: Proving termination with multiset orderings. Commun. ACM 22, 465–476 (1979)
7. Baumgartner, P., Fuchs, A., Tinelli, C.: MELIA – model evolution with linear integer arithmetic constraints (to appear, 2008)
8. Rümmer, P.: A sequent calculus for integer arithmetic with counterexample generation. In: Beckert, B. (ed.) Proceedings, 4th International Verification Workshop. CEUR, vol. 259 (2007), `http://ceur-ws.org/`
9. Schrijver, A.: Theory of Linear and Integer Programming. Wiley, Chichester (1986)
10. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). Journal of the ACM 53, 937–977 (2006)
11. Dowek, G., Hardin, T., Kirchner, C.: Theorem proving modulo. Journal of Automated Reasoning 31, 33–72 (2003)
12. Platzer, A.: Differential dynamic logic for hybrid systems. Journal of Automated Reasoning 41, 143–189 (2008)

13. Stickel, M.E.: Automated deduction by theory resolution. Journal of Automated Reasoning 1, 333–355 (1985)
14. Bürckert, H.J.: A resolution principle for clauses with constraints. In: Stickel, M.E. (ed.) CADE 1990. LNCS, vol. 449, pp. 178–192. Springer, Heidelberg (1990)
15. Korovin, K., Voronkov, A.: Integrating linear arithmetic into superposition calculus. In: Duparc, J., Henzinger, T.A. (eds.) CSL 2007. LNCS, vol. 4646, pp. 223–237. Springer, Heidelberg (2007)

Encoding Queues in Satisfiability Modulo Theories Based Bounded Model Checking*

Tommi Junttila and Jori Dubrovin

Helsinki University of Technology TKK
Department of Information and Computer Science
P.O. Box 5400, FIN-02015 TKK, Finland
Tommi.Junttila@tkk.fi, Jori.Dubrovin@tkk.fi

Abstract. Using a Satisfiability Modulo Theories (SMT) solver as the back-end in SAT-based software model checking allows common data types to be represented directly in the language of the solver. A problem is that many software systems involve first-in-first-out queues but current SMT solvers do not support the theory of queues. This paper studies how to encode queues in the context of SMT-based bounded model checking, using only widely supported theories such as linear arithmetic and uninterpreted functions. Various encodings with considerably different compactness and requirements for available theories are proposed. An experimental comparison of the relative efficiency of the encodings is given.

1 Introduction

Bounded model checking (BMC) [1] is an efficient symbolic model checking technique that has been successfully applied to finding bugs in hardware, software, timed, and hybrid systems. In a recent industrial project we have applied BMC to the analysis of asynchronous, message passing, object oriented systems described in UML [2,3]. Such systems rise naturally e.g. in the context of communication protocol design. The proposed BMC techniques seem to be relatively efficient (especially when the so-called step semantics are applied [3]) and sometimes even complementary to the explicit state methods traditionally used in the analysis of this kind of systems. In [2,3] we use the NuSMV tool [4] as the back-end, and thus the symbolic transition relation is eventually translated ("bit-blasted") into propositional logic and solved with a propositional satisfiability (SAT) solver. Compared to propositional SAT, Satisfiability Modulo Theories (SMT, see e.g. [5,6,7]) offers an attractive framework for solving problems involving constraints over non-Boolean domains such as linear arithmetics over reals or integers, equality with uninterpreted functions (EUF), lists, arrays, and so on. Encouraged by this and the tremendous improvements in the efficiency of SMT solvers during the last few years, we have also implemented and experimented with an SMT-based variant of our UML BMC encoding.

When applying BMC to asynchronous message passing systems, one has to be able to encode *queues* in symbolic form accepted by SMT solvers. Unfortunately, decision

* This work has been financially supported by the Academy of Finland (project 112016), Helsinki Graduate School in Computer Science and Engineering, and Jenny and Antti Wihuri Foundation.

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 290–304, 2008.

procedures for theories of queues, especially with sub-queue relations, can be quite complex (see e.g. [8]) and, to the authors' knowledge, are not currently implemented in any of the state-of-the-art SMT solvers. However, in the context of finding counter-examples to safety properties of message passing systems with BMC, we do not need a full theory of queues but *only* enqueue and dequeue operations.[1] If we wish to check liveness properties with BMC (see e.g. [11]) or use temporal induction [12,13] to prove absence of bugs as well, we also need a predicate that checks whether the contents of a queue are the same at two different time steps. As current SMT solvers do not support these restricted theories of queues either (the theory of recursive data structures [14] and its variants implemented in some SMT solvers are not applicable, as they only support stacks and Lisp-like lists, i.e. last-in-first-out protocol instead of first-in-first-out), we have developed ways to encode queues with other theories. In this paper we study how to do such symbolic queue encodings in the BMC context by using fragments of quantifier-free first order logic supported by the current state-of-the-art SMT solvers. Our goal is to develop queue encodings that (i) are compact (queue encodings can form a significant part of the symbolic transition relation encoding used in BMC), (ii) only require theories that are supported in the current SMT solvers, and (iii) are hopefully efficient to solve. We present several alternative queue encodings that vary considerably in compactness and in what kind of theories they apply; we mainly concentrate on queues with fixed bounded capacity but also present one (very compact) encoding that can handle unbounded queues. We benchmark the proposed encodings by using a simple scalable "stress test" model and some real UML models. Naturally, our queue encodings can also be applied to BMC of any hardware or software system that uses queues, not only message passing protocols.

Related work. Compared to some other theories such as those of arrays or linear arithmetics, there seems to be relatively little work on developing and implementing decision procedures for queues.

In [15], lambda functions are used to describe queues within the context of microprocessor verification. However, the expansion of the lambda functions with beta-substitution, required for getting an SMT problem without lambdas, seems to result in a quadratic blow-up with respect to the BMC bound.

As a part of his thesis [8, Chapter 8], Bjørner develops a decision procedure for queues. However, concatenation of queues as well as sub-queue relations are considered, making the decision procedure rather involved compared to our needs. To our knowledge, it is not implemented in any state-of-the-art SMT solver. Based on the axioms given in [8, Chapter 8], one possibility would be to use the cons/revcons constructors to describe queue contents, then eliminate the revcons constructors by using the axioms, and finally solve the resulting problem with an SMT-solver supporting the theory of recursive data types (e.g. Yices [16] to name just one example of such a solver). However, eliminating the revcons constructors in this way leads to a quadratic explosion in the size of the formula with respect to the BMC bound.

[1] Actually, state machines in UML [9] (as well as in SDL [10]) can also temporarily defer messages; the symbolic translation in [2] can handle this, but due to space limitations we do not consider deferring in this paper.

The queue interface concept we use in this paper is similar to the one proposed in [17,18] for encoding memories in the context of BMC for embedded systems. It seems that queues are easier than memories in this setting as the constraints for memories in [17,18] depend on the BMC bound, making the size of the overall encoding quadratic with respect to the bound; the queue encodings presented in this paper are linear with respect to the bound.

As a final note it should be noticed that some of the underlying high-level concepts in our queue encodings are by no means new. Anyone who has programmed a queue data structure with a programming language such as C or C++ has certainly considered the basic ideas of both the "shifting" and "cyclic" approaches of this paper. But we are not aware of any previous attempts to systematically describe, analyze, and benchmark these approaches in the context of SMT-based BMC. Furthermore, the "linear" approach and the "tag-based" element compression exploit uninterpreted functions for reducing the problem size in, we believe, a novel way.

2 BMC and the Queue Interface

In Model Checking [19], we can consider a system to be composed of a finite vector $s = \langle x_1, \ldots, x_n \rangle$ of typed *state variables*, the set $I \subseteq S$ of *initial states*, and the *transition relation* $R \subseteq S \times S$, where $S = \mathit{domain}(x_1) \times \cdots \times \mathit{domain}(x_n)$ is the set of *states* of the system. A pair $\langle s, s' \rangle$ is in the transition relation iff the system can move from s to s' in one execution step. We will primarily consider checking *invariant properties* of systems and define the set $B \subseteq S$ of *bad states*, in which the invariant is broken. The model checking problem is to determine whether a bad state can be reached from any initial state with a finite number of transitions.

In Bounded Model Checking (BMC), the characteristic functions of the sets I, R, and B are encoded as formulas $I(s)$, $R(s, s')$, and $B(s)$ over vectors of state variables. The question is then whether there is a *bound* $K \geq 0$ and a sequence $s_0, \ldots, s_K$ of states such that $I(s_0) \wedge R(s_0, s_1) \wedge \cdots \wedge R(s_{K-1}, s_K) \wedge B(s_K)$ holds. For a fixed bound, a satisfiability checker, in this case an SMT solver, can decide the existence of a state sequence that satisfies the latter formula and thus show whether a bad state can be reached.

The problem we address is that if one or more of the state variables represent queues, there is no direct way of encoding the queue contents and operations in the language of the currently available SMT solvers. We propose several alternative queue encodings that only use theories that are widely supported by state-of-the-art solvers. We will encapsulate each queue behind a unified interface that allows (limited) access to the contents of the queue, making it possible to switch to a different queue encoding while keeping the encoding of the rest of the system the same. Although we only talk about a single queue in a system, several queues can be handled by simply duplicating the interface and the encoding.

We assume that at each time step, each queue can be the target of at most one enqueue and at most one dequeue operation (both can occur at the same time step provided that the queue is not empty: that is, the element enqueued at a time step cannot be dequeued at the same time step but only later).

The interface consists of two client-controlled Boolean variables, which tell whether a dequeue/enqueue operation is executed, and two data elements: one that tells the client of the queue what the first element in the queue is at the current time step, and another for the element to be appended into the queue in the case the enqueue operation is executed. Furthermore, the client can query whether the queue is empty or full, and whether the contents of the queue at two different time steps match. As mentioned in the introduction, the equality test between time steps is only required if one wishes to check liveness properties or apply temporal induction (see e.g. [11, 12, 13]).

The contents of a queue with element type ELEM can be represented as a variable-length vector of elements, thus $domain(\text{QUEUE}(\text{ELEM})) = domain(\text{ELEM})^*$. In the case of a bounded queue, the length is bounded by the *capacity* Z of the queue.

Formally, the queue interface contains the following terms.

- $empty^t$ and $full^t$ are Boolean formulas that tell whether the queue is empty or full, respectively, at the time step t with $0 \leq t \leq K$. A bounded queue is full iff it contains Z elements. An unbounded queue is never full.
- $firstelem^t$ is of type ELEM and holds the value of the first element in the queue at the time step t. It has a meaningless value if the queue is empty.
- deq^t is a client-controlled Boolean variable that determines whether the first element of the queue is removed when moving to the time step $t + 1$.
- enq^t is a client-controlled Boolean variable that determines whether the element $newelem^t$ is appended to the queue when moving to the time step $t + 1$.
- $newelem^t$ of type ELEM is a client-controlled term, see the previous item.
- $equal^{t,u}$ is a Boolean formula that is true iff the contents of the queue at time steps t and u are the same.

It is assumed that the client of the queue interface never tries to (i) dequeue when the queue is empty, or (ii) enqueue when the queue is full and dequeuing not is taking place at the same time step. That is, the following are assumed to be invariants (i.e. to hold at every time step t):

$$deq^t \Rightarrow \neg empty^t \tag{1}$$

$$enq^t \Rightarrow (\neg full^t \vee deq^t) \tag{2}$$

Formally, the contents of a queue Q evolve in time as follows. Let the contents at the time step t be $Q^t = \langle v_1, v_2, \ldots, v_n \rangle$. Then $firstelem^t = v_1$ and the contents at the next time step are

$$Q^{t+1} = \begin{cases} \langle v_2, \ldots, v_n, newelem^t \rangle & \text{if } \neg empty^t \wedge deq^t \wedge enq^t \\ \langle v_2, \ldots, v_n \rangle & \text{if } \neg empty^t \wedge deq^t \wedge \neg enq^t \\ \langle v_1, \ldots, v_n, newelem^t \rangle & \text{if } \neg deq^t \wedge enq^t \\ \langle v_1, \ldots, v_n \rangle & \text{if } \neg deq^t \wedge \neg enq^t. \end{cases} \tag{3}$$

We point out that the queue interface is not as expressive as a true theory of queues would be. In particular, we cannot define arbitrary relationships between time points, for example, constraining that Q^6 is equal to Q^2 except that the first element has been dequeued. However, from the BMC point of view, arbitrary constraints between queue

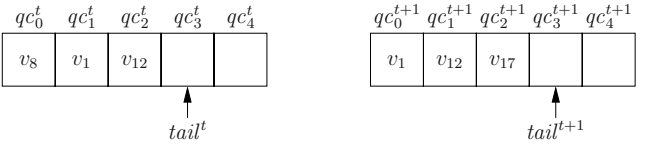


Fig. 1. An illustration of the shifting-based encoding approach when $Q^t = \langle v_8, v_1, v_{12} \rangle$, $deq^t = enq^t = \mathbf{true}$, and $newelem^t = v_{17}$, resulting in $Q^{t+1} = \langle v_1, v_{12}, v_{17} \rangle$

variables are not needed, and it suffices to reason about a non-branching evolution of the contents of the queue. The restrictions on the interface make it possible to design compact encodings to be used especially with BMC.

3 Queue Encodings

In this section we present the three different approaches for encoding queues, called "shifting", "cyclic", and "linear", together with variations within each approach. We analyze the sizes and theory requirements of the alternatives.

3.1 A Shifting-Based Approach

Our first approach, illustrated in Fig. 1, is a straightforward implementation of the BMC semantics of queues given in Eq. (3). It considers bounded queues with at most Z elements and is basically the approach presented in [2] except that only pure FIFO queues are considered here. For each time step t, we introduce a sequence $\langle qc_0^t, \ldots, qc_{Z-1}^t \rangle$ of variables, each of type ELEM. The intuition is that qc_{s-1}^t holds the value of the s:th element in the queue at time step t (the semantics is undefined if there are less than s elements). In addition, we introduce a timed integer variable $tail^t$ that holds the location of the first unused slot in the sequence, i.e. the length of the queue at time step t. Letting s quantify over $\{0, \ldots, Z-1\}$, the definitions for the queue interface variables as well as for updating the queue contents are the following.

$$tail^{t+1} := \begin{cases} deq^t \wedge \neg enq^t & : tail^t - 1 \\ \neg deq^t \wedge enq^t & : tail^t + 1 \\ \mathsf{else} & : tail^t \end{cases} \tag{4}$$

$$empty^t := (tail^t = 0) \tag{5}$$

$$full^t := (tail^t = Z) \tag{6}$$

$$qc_s^{t+1} := \begin{cases} enq^t \wedge \neg deq^t \wedge tail^t{=}s & : newelem^t \\ enq^t \wedge deq^t \wedge tail^t{=}s{+}1 & : newelem^t \\ deq^t & : qc_{s+1}^t \\ \mathsf{else} & : qc_s^t \end{cases} \tag{7}$$

$$firstelem^t := qc_0^t. \tag{8}$$

The notation of Eqs. (4) and (7) should be interpreted as a standard case-expression, i.e. $tail^{t+1} := \mathsf{if}\ deq^t \wedge \neg enq^t\ \mathsf{then}\ tail^t{-}1\ \mathsf{else}\ (\mathsf{if}\ \neg deq^t \wedge enq^t\ \mathsf{then}\ tail^t{+}1\ \mathsf{else}\ tail^t)$.

The term qc_Z^t that appears in (7) in the boundary case $s = Z - 1$ can be taken to have an arbitrary constant value of type ELEM.

As updating the queue contents (Eq. (7)) requires $\mathcal{O}(Z)$ definitions for each time step, the overall size of the encoding with BMC bound K is $\mathcal{O}(K \cdot Z)$.

Equality test. In this approach it is straightforward to check in size $\mathcal{O}(Z)$ whether the contents of the queue are the same at two time steps t and u:

$$equal^{t,u} := (tail^t = tail^u) \wedge \bigwedge_{0 \le s < Z} (s < tail^t) \Rightarrow (qc_s^t = qc_s^u). \tag{9}$$

One-Hot Encoding for the Tail Pointer. The integer tail pointer used in the shifting-based approach above requires that the SMT solver includes a decision procedure for integers with constants and the successor function. We can eliminate this requirement by using a Boolean one-hot encoding for the tail pointer as follows. For each $s \in \{0, 1, \ldots, Z\}$, introduce a timed Boolean variable $tail_s^t$. In any satisfying truth assignment, at each time step t, exactly one of the variables $tail_0^t, \ldots, tail_Z^t$ will be true by construction. Letting s quantify over $\{0, 1, \ldots, Z\}$, the updating of the tail pointer is expressed as

$$tail_s^{t+1} := \begin{cases} enq^t \wedge \neg deq^t & : (s > 0) \wedge tail_{s-1}^t \\ \neg enq^t \wedge deq^t & : (s < Z) \wedge tail_{s+1}^t \\ \mathsf{else} & : tail_s^t \end{cases} \tag{10}$$

where each $(s > 0)$ and $(s < Z)$ is interpreted as a constant **false** or **true** depending on s. Equations (5)–(7) are modified by substituting each equality check of the form $tail^t = c$ with the variable $tail_c^t$.

Although updating the tail pointer now requires $\mathcal{O}(Z)$ definitions (Eq. (10)) instead of one as in the integer case (Eq. (4)), the size of the overall encoding stays in $\mathcal{O}(K \cdot Z)$. The potential benefits of the one-hot encoding are that (i) no additional theories are required by the queue encoding, and (ii) as Boolean SAT solvers (whose search techniques and data structures modern SMT solvers apply) are very efficient, the introduced Boolean constraints are possibly easier to solve for SMT solvers.

Equality test. The predicate for checking the equality of queue contents at two time steps is relatively easy to express also in this encoding. Let $equal^{t,u} := equal_0^{t,u}$, where the auxiliary predicate $equal_0^{t,u}$ is defined as

$$equal_s^{t,u} := \left(tail_s^t \wedge tail_s^u\right) \vee \left((qc_s^t = qc_s^u) \wedge equal_{s+1}^{t,u}\right) \quad \text{for } 0 \le s < Z, \text{ and}$$
$$equal_Z^{t,u} := \left(tail_Z^t \wedge tail_Z^u\right).$$

The size of the $equal^{t,u}$ formula stays the same $\mathcal{O}(Z)$ as in the integer encoded tail pointer case above.

3.2 A Cyclic Approach

We can modify the encoding of Sect. 3.1 by introducing a timed integer variable $head^t$ that tells the position of the first element of the queue. Instead of shifting the entire contents of the queue upon a dequeue operation, we increment $head^t$ by one. This requires

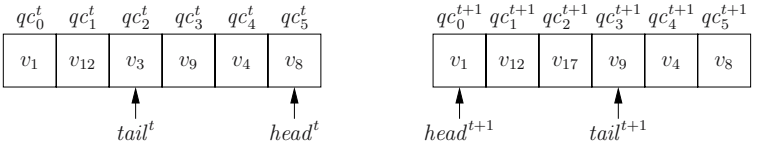


Fig. 2. An illustration of the cyclic encoding approach when $Z = 5$, $Q^t = \langle v_8, v_1, v_{12}\rangle$, $deq^t = enq^t = \mathbf{true}$, and $newelem^t = v_{17}$, resulting in $Q^{t+1} = \langle v_1, v_{12}, v_{17}\rangle$

that the values of $head^t$ and $tail^t$ wrap around at the boundary Z. For notational convenience, we define the terms "successor modulo $Z+1$" and "predecessor modulo $Z+1$" by "$succ(x)$:= if $x = Z$ then 0 else $x+1$" and "$pred(x)$:= if $x = 0$ then Z else $x-1$", respectively. We define

$$head^{t+1} := \text{if } deq^t \text{ then } succ(head^t) \text{ else } head^t \tag{11}$$
$$tail^{t+1} := \text{if } enq^t \text{ then } succ(tail^t) \text{ else } tail^t \tag{12}$$
$$empty^t := (tail^t = head^t) \tag{13}$$
$$full^t := (succ(tail^t) = head^t). \tag{14}$$

There are several approaches for representing the queue contents at each time step, as discussed in the following sub-sections. These vary in compactness and in their requirements for the available decision procedures.

Explicit Contents Representation. As in the shifting-based approach in Sect. 3.1, we introduce a sequence $\langle qc_0^t, \ldots, qc_Z^t\rangle$ of timed variables, each of type ELEM. The queue contents update and the $firstelem^t$ term are written as follows (see Fig. 2).

$$qc_s^{t+1} := \text{if } enq^t \wedge (tail^t{=}s) \text{ then } newelem^t \text{ else } qc_s^t \tag{15}$$

$$firstelem^t := \begin{cases} head^t = 0 & : qc_0^t \\ \quad\vdots & \quad\vdots \\ head^t = Z-1 & : qc_{Z-1}^t \\ \text{else} & : qc_Z^t \end{cases} \tag{16}$$

There are $Z+1$ frame definitions (15), each of constant size. On the other hand, Eq. (16) is of size $\mathcal{O}(Z)$. Thus the size of the overall BMC encoding is $\mathcal{O}(K \cdot Z)$.

Equality test. Perhaps the easiest way of forming the equality predicate $equal^{t,u}$ is to use the one presented below for one-hot encoded head and tail pointers and simply replace each test of the form "$head_c^t$" with "$head^t = c$" and "$tail_c^t$" with "$tail^t = c$".

One-Hot Encoding for Head and Tail. Similarly to Sect. 3.1, we can express the head and tail pointers with Boolean one-hot encoding instead of integers. We only apply this one-hot encoding with the explicit contents representation. For each $s \in \{0, 1, \ldots, Z\}$, introduce timed Boolean variables $head_s^t$ and $tail_s^t$. We replace definitions (11)–(14) with the following for $s = 0, \ldots, Z$.

$$head_s^{t+1} := \text{if } deq^t \text{ then } head_{pred(s)}^t \text{ else } head_s^t \tag{17}$$

$$tail_s^{t+1} := \text{if } enq^t \text{ then } tail_{pred(s)}^t \text{ else } tail_s^t \tag{18}$$

$$empty^t := \bigvee_{0 \le s' \le Z} (head_{s'}^t \wedge tail_{s'}^t) \tag{19}$$

$$full^t := \bigvee_{0 \le s' \le Z} (head_{s'}^t \wedge tail_{pred(s')}^t) \tag{20}$$

Again, the definitions (15) and (16) are modified by substituting each test $head^t = c$ with $head_c^t$ and each $tail^t = c$ with $tail_c^t$.

Compared to the integer case (Eqs. (11)–(14)), maintaining head, tail, empty, and full definitions now requires terms of size $\mathcal{O}(Z)$ instead of $\mathcal{O}(1)$ per time step. However, the total BMC encoding size for K steps stays in $\mathcal{O}(K \cdot Z)$.

Equality test. The predicate for checking the equality of queue contents at two time steps is more cumbersome than in Sect. 3.1 as the head position is now variable. We can define

$$equal^{t,u} := \bigwedge_{0 \le i,j \le Z} (head_i^t \wedge head_j^u) \Rightarrow E_{i,j}^{t,u}, \tag{21}$$

where the $E_{i,j}^{t,u}$ are predicates constrained by

$$E_{i,j}^{t,u} \Leftrightarrow (tail_i^t \wedge tail_j^u) \vee \left(\neg tail_i^t \wedge \neg tail_j^u \wedge (qc_i^t{=}qc_j^u) \wedge E_{succ(i),succ(j)}^{t,u}\right). \tag{22}$$

Intuitively, $E_{i,j}^{t,u}$ is a "suffixes are equal" predicate evaluating to true iff the sequence $\langle qc_i^t, qc_{succ(i)}^t, \ldots, qc_{pred(tail^t)}^t \rangle$ is the same as $\langle qc_j^u, qc_{succ(j)}^u, \ldots, qc_{pred(tail^u)}^u \rangle$. The size of the $equal^{t,u}$ predicate, including the constraints in (22), is $\mathcal{O}(Z^2)$.

UIF-Based Contents Representation. Instead of having a variable qc_s^t to represent the value of the element s at the time step t, we can encode the contents of all elements at a single time step by using an uninterpreted function (UIF). That is, for each time step t, we introduce an UIF $\mathsf{qc}^t : \text{INT} \rightarrow \text{ELEM}$ and rewrite the equations (15) and (16) as

$$\mathsf{qc}^{t+1}(s) := \text{if } enq^t \wedge (tail^t{=}s) \text{ then } newelem^t \text{ else } \mathsf{qc}^t(s) \tag{23}$$

$$firstelem^t := \mathsf{qc}^t(head^t) \tag{24}$$

where s ranges over $\{0, ..., Z\}$. The idea is to reduce the size of the definition of $firstelem^t$ from $\mathcal{O}(Z)$ to a constant. However, the overall encoding size still remains in $\mathcal{O}(K \cdot Z)$ as the frame constraints (23) are essentially the same as in the explicit encoding.

Equality test. In the cyclic approach, we can express the length of the queue with "$len^t := \text{if } head^t \le tail^t \text{ then } tail^t - head^t \text{ else } tail^t + Z + 1 - head^t$". Now we can define the queue contents equality checking predicate as

$$equal^{t,u} := (len^t = len^u) \wedge \bigwedge_{0 \le i < Z} ((i < len^t) \Rightarrow E_i^{t,u}) \tag{25}$$

where $E_i^{t,u} := \left(\mathsf{qc}^t(succ^i(head^t)) = \mathsf{qc}^u(succ^i(head^u))\right)$, and $succ^i(x)$ denotes the nested application of the $succ(x)$-notation i times.

Array-Based Contents Representation. We can avoid writing the $Z+1$ copies of the frame constraint (23) by using an array instead of an UIF to represent the queue contents. The downside is the reliance on the more complex theory of arrays (see e.g. [20]). We denote the operations on arrays by $read(a, i)$, which returns the value at index i in array a, and $write(a, i, v)$, which returns a copy of array a in which the value at index i has been replaced by v. We introduce a timed array variable $qc^t : \text{INT} \rightarrow \text{ELEM}$ that describes the queue contents at time t. The definitions (15) and (16) are replaced with

$$qc^{t+1} := \mathsf{if}\ enq^t\ \mathsf{then}\ write(qc^t, tail^t, newelem^t)\ \mathsf{else}\ qc^t \tag{26}$$

$$firstelem^t := read(qc^t, head^t). \tag{27}$$

That is, only a constant amount of definitions are needed for each time step, meaning that the encoding is independent of the queue capacity Z and the size of the resulting overall BMC encoding is $\mathcal{O}(K)$.

Equality test. Unfortunately the compactness of the array-based contents representation does not seem to extend to equality checking. The most compact way we have found for expressing $equal^{t,u}$ in this setting is essentially the one for UIF-based contents representation given in Eq. (25). The only change is to replace each equality test of the form $\mathsf{qc}^t(i) = \mathsf{qc}^u(j)$ with $read(qc^t, i) = read(qc^u, j)$.

3.3 A Linear Approach

We next show a very compact encoding approach exploiting uninterpreted functions and a small fragment of linear arithmetic. The resulting encoding has only a *constant* amount of constraints per time step. A drawback is that, like the UIF- and array-based contents representation approaches above, it requires theory combination: if handling of queue elements otherwise requires a decision procedure for a theory T, then the combination of T with the theory of "EUF + integer offsets" (see [21] for an efficient decision procedure for this theory) is required after introducing the queue constraints.

The basic idea, illustrated in Fig. 3, is very simple: we have a *single*, *infinite* array *common to all time steps* in which the queue progresses as a sliding window. For each time step t, we introduce two integer variables, $head^t : \text{INT}$ and $tail^t : \text{INT}$. The contents of the queue at the time step are the array elements from the index $head^t$ to $tail^t - 1$. When an element is removed from the queue, the $head^t$ variable is incremented by one. Similarly, when an element is inserted in the queue, it is written to the array at index $tail^t$, after which the tail index is incremented by one. Thus each array index is *written at most once*. This allows us to capture the contents of the queue by using an UIF $\mathsf{qc} : \text{INT} \rightarrow \text{ELEM}$. In contrast to the UIF-based contents encoding in Sect. 3.2, the UIF is not time-dependent but shared across all time steps. We define

$$head^{t+1} := \mathsf{if}\ deq^t\ \mathsf{then}\ head^t + 1\ \mathsf{else}\ head^t \tag{28}$$

$$tail^{t+1} := \mathsf{if}\ enq^t\ \mathsf{then}\ tail^t + 1\ \mathsf{else}\ tail^t \tag{29}$$

$$empty^t := (head^t = tail^t) \tag{30}$$

$$full^t := (tail^t = head^t + Z) \tag{31}$$

$$firstelem^t := \mathsf{qc}(head^t) \tag{32}$$

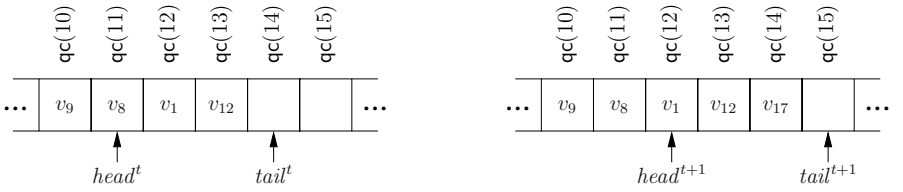


Fig. 3. An illustration of the linear encoding approach when $Q^t = \langle v_8, v_1, v_{12} \rangle$, $deq^t = enq^t = \mathbf{true}$, and $newelem^t = v_{17}$, resulting in $Q^{t+1} = \langle v_1, v_{12}, v_{17} \rangle$

and constrain that

$$enq^t \Rightarrow (\mathsf{qc}(tail^t) = newelem^t). \tag{33}$$

Only a constant amount of terms is needed for each time step, and thus the size of the overall encoding for K time steps is $\mathcal{O}(K)$. The size is the same as that of the bounded cyclic, array-based encoding. The relative benefit is that an easier theory (EUF + integer offsets [21]) is applied and that unbounded queues can be supported (as explained below).

Equality test. The equality checking predicate can be expressed in size $\mathcal{O}(Z)$. Let $len^t := tail^t - head^t$ and define $equal^{t,u} := equal_0^{t,u}$, where

$$equal_s^{t,u} := \big(len^t{=}s \wedge len^u{=}s\big) \vee \big((\mathsf{qc}(head^t{+}s){=}\mathsf{qc}(head^u{+}s)) \wedge equal_{s+1}^{t,u}\big) \tag{34}$$

for $0 \leq s < Z$, and

$$equal_Z^{t,u} := \big(len^t = Z \wedge len^u = Z\big). \tag{35}$$

Unbounded Queues. The linear approach can be modified to allow encoding of *unbounded* queues. Simply replace Eq. (31) above with

$$full^t := \mathbf{false}. \tag{36}$$

The size of the encoding stays in $\mathcal{O}(1)$ per time step.

Equality test. The queue contents equality comparison is similar to that of the bounded case except that the size of the queue at a time step t is now bounded above by $t + M$ instead of the queue capacity Z, where M is the number of elements in queue at the first time step 0. That is, assuming $t < u$, the equality checking predicate $equal^{t,u}$ is the same as in the bounded linear case considered above except that Z is replaced with the constant $t + M$ in Eqs. (34) and (35). The worst case size of $equal^{t,u}$ is thus $\mathcal{O}(K)$. This is a drawback as in BMC for liveness properties we usually have to apply at least K such predicates and thus the overall BMC encoding becomes at least quadratic in the bound.

4 Tag-Based Tuple Element Compression

It is often the case that a queue does not contain scalar values, but *tuples* of values. In the context of UML model checking [3], this happens when the messages in the input

queues of state machines are composed of both a signal identifier and some parameter values associated with the signal. Some SMT solvers, such as Yices [16] and Z3 [22], have a direct support for tuples and thus one can simply define ELEM to be tuple type and use the presented queue encodings unmodified. We next consider solutions for the case that tuples are not supported.

The straightforward way to handle tuple types is to split them into individual parts. For example, if ELEM $=$ TUPLE(REAL, INT), then an uninterpreted function qc^t : INT $\rightarrow$ ELEM appearing in the UIF-based contents representation scheme (Sect. 3.2) is encoded as two UIFs qc_1^t : INT $\rightarrow$ REAL and qc_2^t : INT $\rightarrow$ INT, and a constraint $\mathsf{qc}^t(x) = \mathsf{qc}^t(y)$ becomes $\mathsf{qc}_1^t(x){=}\mathsf{qc}_1^t(y) \wedge \mathsf{qc}_2^t(x){=}\mathsf{qc}_2^t(y)$. This transformation can be applied to all queue encodings of Sect. 3. If the element type ELEM is a tuple with A parts, then the variables $\mathit{firstelem}^t$, $\mathit{newelem}^t$, qc_s^t, and qc^t, and the UIFs qc^t and qc need to be duplicated A times together with the constraints involving those variables and UIFs. This increases the sizes of all encodings by a factor of A in the $\mathcal{O}$-notation. We will call the result the *duplicating* tuple encoding.

The alternative we propose is a *tag-based* encoding that avoids storing tuple values in the queue and moving them across time steps. Instead, each enqueued tuple is associated with a *tag*, e.g. a single integer value, which is stored in the queue. Upon dequeuing, the tag is decoded back into a tuple. The scheme can be efficiently implemented using UIFs as follows. Assume that ELEM $=$ TUPLE$(T_1, \ldots, T_A)$ for some types T_i. We define a scalar type TAG that has to have a domain large enough to hold the possible element values, i.e. $|\mathit{domain}(\text{TAG})| \geq |\mathit{domain}(\text{ELEM})|$ should hold. We define time-independent UIFs decode_i : TAG $\rightarrow T_i$ for each $1 \leq i \leq A$ that are used to interpret the tags as tuple parts, and construct a queue with element type TAG using one of the encodings presented in the previous section. We rename the terms $\mathit{firstelem}^t$ and $\mathit{newelem}^t$ of the interface of the queue as $\mathit{firsttag}^t$ and newtag^t, respectively, and hide them from the client. Instead, the client will see the terms

$$\mathit{firstelem}_i^t := \mathsf{decode}_i(\mathit{firsttag}^t) \tag{37}$$

$$\mathit{newelem}_i^t := \mathsf{decode}_i(\mathit{newtag}^t) \tag{38}$$

for each $1 \leq i \leq A$ as part of the queue interface. Except for the queue equality predicate $\mathit{equal}^{t,u}$ discussed below, this additional level of abstraction does not affect the semantics of the queue. Note that only the decode functions together with the definitions (37) and (38) are duplicated A times, while the internals of the queue only deal with scalar values. Thus when the tag-based encoding is applied, the size of the shifting-based approach as well as that of the cyclic approach with explicit and UIF-based contents representation drop from $\mathcal{O}(K \cdot Z \cdot A)$ to $\mathcal{O}(K \cdot (Z + A))$. The size of the array-based contents representation stays in $\mathcal{O}(K \cdot A)$ but requires only one array variable instead of A per time step; this is, in theory, beneficial as the theory of equality with UIFs required by tags is much easier to decide than the theory of arrays.

As tuples with same values can be assigned to different tags, the equality checking predicate $\mathit{equal}^{t,u}$ needs special treatment. In $\mathit{equal}^{t,u}$, every equality comparison between tag values has to be expanded; for instance, the comparison $qc_s^t = qc_s^u$ in Eq. (9) has to be rewritten as $\bigwedge_{1 \leq i \leq A} (\mathsf{decode}_i(qc_s^t) = \mathsf{decode}_i(qc_s^u))$. Thus for the equality checking part of the encoding, tags do not help to compress the size of the encoding.

Table 1. Comparison of the approaches on the single queue stress test with a scalar integer element. The numbers show the largest bound that was solved within ten minutes by the Z3 solver.

	shifting		cyclic				linear	unbounded
			explicit		uif	array		
Z	int. tail	one-hot tail	int. tail	one-hot tail				
2	28	79	35	**92**	24	26	13	12
5	20	33	28	**42**	15	26	12	
10	19	25	20	**31**	14	27	12	
50	19	25	17	**31**	12	25	11	

5 Experiments

We now provide an experimental comparison of the proposed approaches. We have developed a prototyping tool called PySMT for constructing and solving SMT problems in the Python programming language. It (i) has an API for constructing the problems, (ii) can translate the problems into the native input language of several SMT solvers and also (to some extent) to the SMT-LIB format, and (iii) can also execute the solver binary and (to a quite limited extent) parse the result so that it can be queried by using the API. We have implemented the proposed queue encoding approaches on top of PySMT and give some preliminary experimental results below. The scripts and models for the experiments are available at `http://www.tcs.hut.fi/~tjunttil/experiments/LPAR2008-SMT`.

5.1 Single Queue Stress Test

First, we try to test the efficiency of the queue encodings in isolation by constructing very simple BMC problems consisting of one queue only. In this problem, (i) whether an enqueue or a dequeue operation is applied at time step t is unconstrained, meaning that all possible enqueue/dequeue sequences are considered, (ii) each enqueued element is either an integer or a tuple of integers, each constrained to have a value greater than some positive constant, and (iii) the (valid) property to be checked is that an element with a negative integer value is never dequeued. For each time step, we set the run time limit to ten minutes and report the time spent in the solver. Problem generation time is not included as our generator script has, we believe, unessential inefficiencies.

Scalar Elements. To isolate the core queue constraints from the constraints needed to represent tuple elements, we first compare the encoding approaches in the case of scalar elements. Table 1 shows the results for different queue lengths when Z3 (version 1.2) [22] is used as the SMT solver. In addition, Fig. 4(a) shows a more detailed view when the queue size Z is five. The results show that there are dramatic differences in the performance of different approaches and contents encoding schemes. Unfortunately, the more compact and elegant ones, namely the linear approach and the cyclic approach with array-based contents representation, are not performing well. Instead, the encodings applying fewest theories, i.e. the cyclic and shifting approaches with one-hot

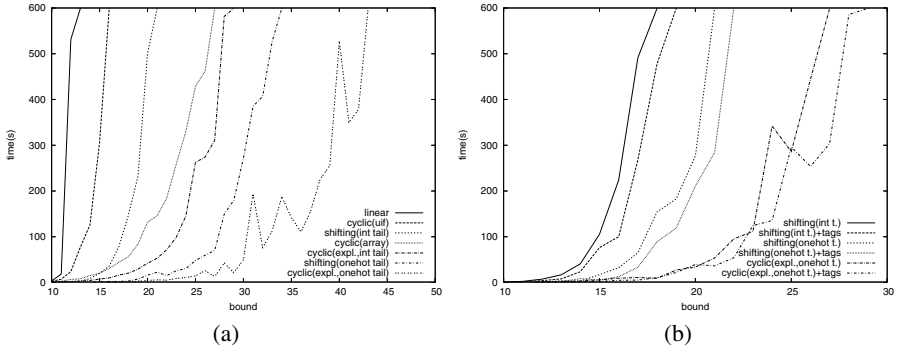
encoded head and tail pointers seem to be the best choices in general. We also ran experiments with the Yices SMT solver [16] and obtained similar results, except that the integer encoded head and tail pointers seem to perform as well as the one-hot encoded.

Tuple Elements. We also benchmarked the encoding approaches when queue elements are tuples of integers. As expected, the result is that the problems become harder when tuples contain more parts. For instance, consider the cyclic approach with explicit contents representation and one-hot encoded head and tail pointers. With scalar element (tuple with one part) the behavior is shown as the rightmost curve in 4(a) while the second rightmost curve in Fig. 4(b) shows the same with a tuple of five parts: the largest bound solved within ten minutes drops from 42 to 27.

The second observation is that the tag-based tuple encoding is almost universally a few times more efficient in terms of running time than the duplicating tuple encoding. Figure 4(b) shows a comparison of the tuple encodings for three different queue encodings approaches; these plots represent typical behavior in this benchmark set. We also experimented with the direct tuple type support of Z3; it seems to provide similar performance as our tag-based encoding. Again, comparable results were obtained when Yices was used as the solver instead of Z3.

5.2 Bounded Model Checking of UML Models

We have also benchmarked the queue encodings in a more realistic bounded model checking context. We analyzed some UML models by using the symbolic encoding described in [2,3]. Instead of using NuSMV, we translate the BMC problems into SMT problems and use Yices (version 1.0.11) [16] to solve them. The results are shown in Table 2, the numbers give the cumulative time (in seconds) used by the SMT solver when solving all the problems from bound 0 to the bound $|\text{cex}|$ where a counter-example to the analyzed property is found. The queue size for the bounded queue encodings was set to ten; "dstep" ("interl.", resp.) denotes that the dynamic step (interleaving, resp.) semantics (see [3]) was applied. The use of tags to represent tuple queue elements seems



(a) (b)

Fig. 4. Comparison of some approaches with Z3 as the solver. (a) Scalar queue element, $Z = 5$. (b) Tuple queue element with 5 parts, $Z = 5$.

Table 2. BMC of some UML models using Yices as the solver

model	|cex|	linear unbounded	shifting	shifting one-hot	cyclic explicit	cyclic one-hot	cyclic UIF
`giop`, dstep	8	21.31	23.93	28.99	296.77	78.98	137.33
(with tags)			24.18	24.52	29.58	25.67	24.44
`giop`, interl.	14	1986.07	1599.10	1250.13	>1h	>1h	2902.11
(with tags)			1084.25	860.11	849.31	1244.03	1088.80
`travel`, interl.	15	1.86	2.54	2.18	3.16	2.93	3.17
(with tags)			2.45	2.09	2.87	2.83	3.26
`mtravel`, dstep	11	3.77	5.43	4.27	7.45	5.74	7.02
(with tags)			3.99	3.79	4.58	4.32	4.82

to play a much bigger role than the encoding approach in this practical setting; they provide a substantial performance gain especially when analyzing the `giop` model having tens of message parameters and thus wide tuples in queues. With the other models having no or only few parameters, the performance gain is non-existent or small; in addition, the choice of the encoding approach does not seem to make much difference on these models. The reason for this is probably that the applied bounds are relatively small and parts other than the queue encoding dominate the search space of the SMT problem.

6 Conclusions

We have presented and experimentally evaluated different quantifier-free SMT encodings for queues in the context of bounded model checking. The presented encodings vary significantly in compactness and the theories they require the SMT solver to implement. Our preliminary experimental results show that the most compact encodings do not necessarily perform best, even when they involve no complex theories such arrays but only equality with uninterpreted functions and integer offsets. On the contrary, it seems that it may be worthwhile to use more space by booleanizing integer head and tail pointers so that the encoding becomes essentially propositional, the only theory atoms being equality tests between elements that are stored in the queue. The proposed method for compressing tuple elements with the use of tags and uninterpreted decode functions yields a relatively consistent and often significant speed-up in our experiments. The most obvious future work is of course to develop and implement decision procedures for theories of queues. The encoding approaches presented in this paper form a natural base when evaluating their performance in the BMC context.

References

1. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
2. Dubrovin, J., Junttila, T.: Symbolic model checking of hierarchical UML state machines. In: ACSD 2008, pp. 108–117. IEEE Press, Los Alamitos (2008)

3. Dubrovin, J., Junttila, T., Heljanko, K.: Symbolic step encodings for object based communicating state machines. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 96–112. Springer, Heidelberg (2008)
4. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV version 2: An opensource tool for symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 359–364. Springer, Heidelberg (2002)
5. de Moura, L.M., Dutertre, B., Shankar, N.: A tutorial on satisfiability modulo theories. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 20–36. Springer, Heidelberg (2007)
6. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). Journal of the ACM 53(6), 937–977 (2006)
7. Bozzano, M., Bruttomesso, R., Cimatti, A., Junttila, T.A., van Rossum, P., Schulz, S., Sebastiani, R.: MathSAT: Tight integration of SAT and mathematical decision procedures. Journal of Automated Reasoning 35(1–3), 265–293 (2005)
8. Bjørner, N.S.: Integrating Decision procedures for Temporal Verification. PhD thesis, Stanford University (1998)
9. OMG: UML 2.0 superstructure specification (2005), http://www.omg.org
10. International Telecommunication Union Geneva, Switzerland: Recommendation Z.100 (03/93) - CCITT specification and description language (SDL) (1993)
11. Biere, A., Heljanko, K., Junttila, T.A., Latvala, T., Schuppan, V.: Linear encodings of bounded LTL model checking. Logical Methods in Computer Science 2(5) (2006)
12. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: Johnson, S.D., Hunt Jr., W.A. (eds.) FMCAD 2000. LNCS, vol. 1954, pp. 108–125. Springer, Heidelberg (2000)
13. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. In: BMC 2003. Electronic Notes in Theoretical Computer Science, vol. 89, pp. 541–638. Elsevier, Amsterdam (2003)
14. Oppen, D.C.: Reasoning about recursively defined data structures. Journal of the ACM 27(3), 403–411 (1980)
15. Lahiri, S.K., Seshia, S.A., Bryant, R.E.: Modeling and verification of out-of-order microprocessors in UCLID. In: Aagaard, M.D., O'Leary, J.W. (eds.) FMCAD 2002. LNCS, vol. 2517, pp. 142–159. Springer, Heidelberg (2002)
16. Dutertre, B.: System description: Yices 1.0.10. SMT-COMP 2007 tool description paper (2007), http://www.smtcomp.org/2007/participants.shtml
17. Ganai, M.K., Gupta, A., Ashar, P.: Efficient modeling of embedded memories in bounded model checking. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 440–452. Springer, Heidelberg (2004)
18. Ganai, M.K., Gupta, A., Ashar, P.: Verification of embedded memory systems using efficient memory modeling. In: DATE 2005, pp. 1096–1101. IEEE Computer Society, Los Alamitos (2005)
19. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge (1999)
20. Stump, A., Barrett, C.W., Dill, D.L., Levitt, J.R.: A decision procedure for an extensional theory of arrays. In: LICS 2001, pp. 29–37. IEEE Computer Society, Los Alamitos (2001)
21. Nieuwenhuis, R., Oliveras, A.: Fast congruence closure and extensions. Information and Computation 205, 557–580 (2007)
22. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)

On Bounded Reachability of Programs with Set Comprehensions

Margus Veanes[1] and Ando Saabas[2,⋆]

[1] Microsoft Research, Redmond, WA, USA
margus@microsoft.com
[2] Institute of Cybernetics, Tallinn University of Technology, Tallinn, Estonia
ando@cs.ioc.ee

Abstract. We analyze the bounded reachability problem of programs that use abstract data types and set comprehensions. Such programs are common as high-level executable specifications of complex protocols. We prove decidability and undecidability results of restricted cases of the problem and extend the Satisfiability Modulo Theories approach to support analysis of set comprehensions over tuples and bag axioms. We use the Z3 solver for our implementation and experiments, and we use AsmL as the modeling language.

1 Introduction

Programs that use high-level data types are commonly used to describe executable specifications [22] in form of so called *model programs*. An important and growing application area in the software industry is the use of model programs for specifying and documenting expected behavior of application-level network protocols [14]. Model programs typically use abstract data types such as sets and maps, and comprehensions to express complex state updates. Correctness assumptions about the model are usually expressed through state invariants. An important problem is to validate a model prior to its use as an oracle or final specification. One approach is to use Satisfiability Modulo Theories or SMT to perform bounded reachability analysis or bounded model-checking of model programs [36]. The use of SMT solvers for automatic software analysis has recently been introduced [1,11] as an extension of SAT-based bounded model checking [5]. The SMT based approach makes it possible to deal with more complex background theories. Instead of encoding the verification task of a sequential program as a propositional formula the task is encoded as a quantifier free formula. The decision procedure for checking the satisfiability of the formula may use combinations of background theories [29].

The main contribution of this paper is a characterization of the decidable and undecidable cases of the bounded reachability problem of model programs. We show in Section 3 that already the single step reachability problem is undecidable if a single set-valued parameter is allowed. In Section 4 we show that

⋆ Part of this work was done during an internship at Microsoft Research, Redmond.

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 305–317, 2008.

the bounded reachability problem remains decidable provided that all parameters have basic (non-set valued) types. This result is orthogonal to the decidable fragment of bounded reachability of model programs that use the array property fragment [8]. In Section 5 the paper extends the work started in [36] through improved handling of quantifier instantiation and extended support for background axioms to support for example bag or multi-set axioms. We use the SMT solver Z3 [10] for our experiments and we use AsmL [16] as the modeling language. Related work is discussed in Section 6.

2 Model Programs and Bounded Reachability

In this section we define some background material related to model programs, in order to make the paper self-contained. A more thorough exposition can be found in [36].

Model programs. The main use of model programs is as high-level specifications in model-based testing tools such as Spec Explorer [37] and NModel [30]. In Spec Explorer, one of the supported input languages is the abstract state machine language AsmL [16]. AsmL is used in this paper as the concrete specification language for update rules that correspond to basic ASMs [15].

We let Σ denote the overall signature of function symbols. Part of Σ is denoted by Σ^{var} and contains nullary function symbols whose interpretation may vary from state to state, called *state variables*. The remaining part Σ^{static} contains symbols whose interpretation is fixed by the background theory. A ground term over Σ^{static} is called a *value term*. A subset of Σ^{static}, denoted by Σ^{acts} are free constructors called *action symbols*. Given an action symbol f, an *action* or *f-action* is a value term whose function symbol is f.

For all action symbols f with arity $n \geq 0$, and all i, $1 \leq i \leq n$, there is a unique *parameter variable* (not in Σ^{var}) denoted by f_i. We write Σ_f for $\{f_i\}_{1 \leq i \leq n}$. Note that if $n = 0$ then $\Sigma_f = \emptyset$.

Definition 1. A *model program* P is a tuple (V_P, A_P, I_P, R_P), where

- V_P is a finite set of *state variables*, let Σ_P denote $\Sigma^{\text{static}} \cup V_P$;
- A_P is a finite set of *action symbols*;
- I_P is a formula over Σ_P, called the *initial state condition*;
- R_P is a family $\{R_P^f\}_{f \in A_P}$ of *action rules* $R_P^f = (G_P^f, U_P^f)$, where
  - G_P^f is a quantifier free formula over $\Sigma_P \cup \Sigma_f$ called the *guard*;
  - U_P^f, called the *update rule*, is a block $\{v := t_v^f\}_{v \in V_P^f}$ of *assignments* where t_v^f is a term over $\Sigma_P \cup \Sigma_f$ and $V_P^f \subseteq V_P$.

This definition is a variation of model programs that syntactically restricts the update rules to be block assignments. This restriction is not a true limitation because *if-then-else* terms are allowed and nondeterministic choices can be encoded as branching based on action parameter values (i.e. the choices are made

explicit). We often say *action* to also mean an action rule or an action symbol, if the intent is clear from the context.

In general, model programs can have a rich background theory, including the theory of maps. In the following example, the fragment of interest is the so-called *array theory* fragment where all map sorts have domain sort $\mathbb{Z}$ and the theory of $\mathbb{Z}$ is Presburger arithmetic.

Example 1 (Credits). The following model program is written in AsmL. It specifies how a client and a server need to use message ids, based on a sliding window protocol. It models part of the credits-algorithm in the SMB2 [34] protocol.

```
var window as Set of Integer = {0}
var maxId as Integer = 0
var requests as Map of Integer to Integer = {->}

[Action("Req(_,m,c)")] Req(m as Integer, c as Integer)
  require m in window and c > 0
  requests := Add(requests,m,c)
  window := window difference {m}

[Action("Res(_,m,c,_)")] Res(m as Integer, c as Integer)
   require m in requests
   require requests(m) >= c
   require c >= 0
   window := window union {maxId + i | i in {1..c}}
   requests := RemoveAt(requests,m)
   maxId := maxId + c

[Invariant] ClientHasEnoughCredits()
   require requests = {->} implies window <> {}
```

The *Credits* model program illustrates a typical use of model programs as protocol-specifications. Actions use parameters, maps and sets are used as state variables and a comprehension expression is used to compute a set. (Since the domain of the maps and sets is $\mathbb{Z}$, the example is in the array theory fragment.) Each action has a guard and an update rule given by a basic ASM. For example, the guard of the `Req` action requires that the id of the message is in the current window of available ids and that the number of credits that the client requests from the server is positive. The state invariant associated with the model program is that the client must not starve, i.e. there should always be a message id available at some point, so that the client can issue new requests.

Let P be a fixed model program. A *P-state* is a mapping of V_P to values.[1] Given an action $a = f(a_1, \ldots, a_n)$, let θ_a denote the parameter assignment $\{f_i \mapsto a_i\}_{1 \le i \le n}$. Given a P-state S, an extension of S with the parameter assignment θ is denoted by $(S; \theta)$.

Let S be a P-state, an f-action a is *enabled* in S if $(S; \theta_a) \models G_P^f$ (where $\models$ is the standard satisfaction relation of first-order logic). The action a *causes a transition from S to S'*, where

$$S' = \{v \mapsto t_v^{f(S;\theta_a)}\}_{v \in V_P^f} \cup \{v \mapsto v^S\}_{v \in V_P \setminus V_P^f}.$$

[1] More precisely, this is the foreground part of the state, the background part is the canonical model of the background theory $\mathcal{T}$.

A *labeled transition system* or *LTS* is a tuple $(\mathcal{S}, \mathcal{S}_0, L, T)$, where $\mathcal{S}$ is a set of *states*, $\mathcal{S}_0 \subseteq \mathcal{S}$ is a set of *initial states*, L is a set of labels and $T \subseteq \mathcal{S} \times L \times \mathcal{S}$ is a *transition relation*.

Definition 2. Let P be a model program. The *LTS of* P, denoted by $[\![P]\!]$ is the LTS $(\mathcal{S}, \mathcal{S}_0, L, T)$, where $\mathcal{S}_0$, is the set of all P-states s such that $s \models I_P$; L is the set of all actions over A_P; T and $\mathcal{S}$ are the least sets such that, $\mathcal{S}_0 \subseteq \mathcal{S}$, and if $s \in \mathcal{S}$ and there is an action a that causes a transition from s to s' then $s' \in \mathcal{S}$ and $(s, a, s') \in T$.

A *run* of P is a sequence of transitions $(s_i, a_i, s_{i+1})_{i<\kappa}$ in $[\![P]\!]$, for some $\kappa \leq \omega$, where s_0 is an initial state of $[\![P]\!]$. The sequence $(a_i)_{i<\kappa}$ is called an (*action*) *trace* of P. The run or the trace is *finite* if $\kappa < \omega$.

Bounded reachability of model programs. Let P be a model program and let φ be a Σ_P-formula. The main problem we are addressing is whether φ is reachable in P within a given bound.

Definition 3. Given φ and $k \geq 0$, φ *is reachable in* P *within* k *steps*, if there exists an initial state s_0 and a (possibly empty) run $(s_i, a_i, s_{i+1})_{i<l}$ in P, for some $l \leq k$, such that $s_l \models \varphi$. If so, the action sequence $\alpha = (a_i)_{i<l}$ is called a *reachability trace for* φ and s_0 is called an *initial state for* α.

Note that, given a trace α and an initial state s_0 for it, the state where the condition is reached is reproducible by simply executing α starting from s_0. This provides a cheap mechanism to check if a trace produced by a solver is indeed a witness. In a typical model program, the initial state is uniquely determined by an initial assignment to state variables, so the initial state witness is not relevant.

The *bounded reachability formula* for a given model program P, step bound k and reachability condition φ is:

$$\begin{aligned} \mathit{Reach}(P, \varphi, k) &\stackrel{\text{def}}{=} I_P \wedge \Big(\bigwedge_{0 \leq i < k} P[i]\Big) \wedge \Big(\bigvee_{0 \leq i \leq k} \varphi[i]\Big) \\ P[i] &\stackrel{\text{def}}{=} \bigvee_{f \in A_P} \Big(\mathit{action}[i] = f(f_1[i], \ldots, f_n[i]) \wedge G_P^f[i] \\ &\qquad \bigwedge_{v \in V_P^f} v[i+1] = t_v^f[i] \bigwedge_{v \in V_P \setminus V_P^f} v[i+1] = v[i]\Big) \end{aligned}$$

where an expression $E[i]$ denotes E where each state variable and parameter variable has been given index i if $i > 0$. A *skip* action has the action rule $(\mathit{true}, \emptyset)$. We use the following Theorem from [36].

Theorem 1. *Let P be a model program that includes a skip action, $k \geq 0$ a step bound and φ a reachability condition. Then $\mathit{Reach}(P, \varphi, k)$ is satisfiable if and only if φ is reachable in P within k steps. Moreover, if M satisfies $\mathit{Reach}(P, \varphi, k)$, let $M_0 = \{v \mapsto v^M\}_{v \in V_P}$, let $a_i = \mathit{action}[i]^M$ for $0 \leq i < k$, and let α be the sequence $(a_i)_{i<k}$. Then α is a reachability trace for φ and M_0 is an initial state for α.*

3 One Step Reachability

The bounded reachability problem of model programs is undecidable in the general case. In this section we pin down various minimal cases of the undecidability with respect to certain background theories. In all cases it is enough to restrict the reachability bound and the number of action symbols to 1, i.e. the undecidability arises already using a single step and a single action symbol. We call it the *one step reachability problem*. In Section 4 we argue that these undecidable cases are minimal in some sense.

First, we define a theory $TS(\mathcal{A})$ that extends a given theory $\mathcal{A}$ (for example Presburger arithmetic) with *tuples* and *sets*. It is assumed that the language of $\mathcal{A}$ does not include the new symbols. It is convenient to restrict the set of all possible expressions of $TS(\mathcal{A})$ to a set of well-formed expressions that are shown in Figure 1. When considering a formula of $TS(\mathcal{A})$ as defined in Figure 1, it is assumed that by default all set variables are *existentially quantified*, i.e. have an outermost existential quantifier. We write $TS(\mathcal{A})$ both for the class of expressions as defined in Figure 1, as well as the axioms of $TS(\mathcal{A})$.

The axioms of $TS(\mathcal{A})$ include the axioms of $\mathcal{A}$, the axioms for tuples stating that for each arity k the k-tuple constructor is a free constructor, axioms for set union, set intersection, element-of relation, subset relation, and the extensionality axiom for sets. Given a model $\mathfrak{A}$ of $TS(\mathcal{A})$, i.e., a structure $\mathfrak{A}$ in the language of $TS(\mathcal{A})$ that is a model of the axioms of $TS(\mathcal{A})$, the *comprehension term* $s = \{t(\overline{x}) \mid_{\overline{x}} \varphi(\overline{x})\}$, where t and φ may include parameters, has the interpretation $s^{\mathfrak{A}}$ in $\mathfrak{A}$ such that $\mathfrak{A} \models \forall y(y \in s^{\mathfrak{A}} \leftrightarrow \exists \overline{x}(t(\overline{x}) = y \wedge \varphi(\overline{x})))$ which is well-defined due to the extensionality axiom: $\forall v\, w(\forall y(y \in v \leftrightarrow y \in w) \rightarrow v = w)$.

Example 2. Let $\mathcal{P}$ be Presburger arithmetic. The following is a *range expression*, in $TS(\mathcal{P})$: $\{z \mid x \leq z \wedge z \leq y\}$ where we omit the z from $|_z$. We often use the abbreviation $\{x..y\}$ for a range from x to y. The following is a *direct product* $v \times w$ between two sets v and w: $\{\langle x, y\rangle \mid x \in v \wedge y \in w\}$.

Note that, not all well-formed $TS(\mathcal{P})$ expressions can be used in a model program, in a model program all expressions are quantifier free and each set comprehension variable has a finite range.

Theorem 2. *One can effectively associate a deterministic 2-register machine M with a formula $halts_M(m, n)$ in $TS(\mathcal{P})$ with integer parameters m and n, such that M halts on (m, n) if and only if $halts_M(m, n)$ holds.*

Proof (Outline). Let $STEP_M(\langle i, m, n\rangle, \langle i', m', n'\rangle)$ be the Presburger program formula for M as defined in [6, Theorem 2.1.15], where i, m, n and i', m', n' denote the current and the next configuration of the 2-register machine. Let $1 \ldots k$ be the instructions of M and assume that M is such that the initial instruction is 1 and the final instruction is $k > 1$ and when the final instruction is reached then both registers are zero. Let $halts_M$ be the following formula:

$$halts_M(m, n) \stackrel{\text{def}}{=} \exists s \exists l(valid_M(m, n, s, l))$$

$$
\begin{array}{ll}
\textit{Basic elements}: & E ::= T_{\mathcal{A}} \mid \langle E, \ldots, E\rangle \mid \pi_i(E) \mid x \mid \mathit{ite}(F, E, E) \\
\textit{Sets of basic elements}: & S ::= \{E \mid_{\overline{x}} F\} \mid \emptyset \mid S \cup S \mid S \cap S \mid S \setminus S \mid v \mid \mathit{ite}(F, S, S) \\
\textit{Formulas}: & F ::= F_{\mathcal{A}} \mid \neg F \mid F \wedge F \mid F \vee F \mid \forall x\, F \mid \exists x\, F \mid \\
 & \quad E = E \mid S \subseteq S \mid S = S \mid E \in S
\end{array}
$$

Fig. 1. Well-formed expressions in $TS(\mathcal{A})$. The theory $\mathcal{A}$ has terms $T_{\mathcal{A}}$ and Formulas $F_{\mathcal{A}}$. It is assumed that all terms in $T_{\mathcal{A}}$ have sort $\mathbb{A}$. Set variables are denoted by v and basic variables (tuple variables or variables of sort $\mathbb{A}$) are denoted by x. The grammar omits sorts (type annotations) for ease of readability, but it is that of standard many-sorted first order logic. For example in a set operation term $s_1 \diamond s_2$, it is assumed that both s_1 and s_2 have the same sort (so sets contain only homogeneous elements), in an element-of atom $t \in s$ it is assumed that if the sort of t is σ then the sort of s is $\{\sigma\}$, a tuple (t_1, t_2) has the sort $\sigma_1 \times \sigma_2$ provided that t_i has sort σ_i, etc.

```
type Config = (Integer, Integer, Integer)
steps as Set of (Integer,Config,Config)
length as Integer
[Action] halts_M(m as Integer, n as Integer)
  require valid_M(m, n, steps, length)
```

Fig. 2. Model program P_M

$$
\begin{array}{l}
\mathit{valid}_M(m, n, s, l) \stackrel{\text{def}}{=} \\
\quad s = \{\langle j, x, y\rangle \mid \langle j, x, y\rangle \in s \wedge \mathit{STEP}_M(x, y) \wedge 1 \leq j \wedge j \leq l\} \wedge \\
\quad \{\langle \pi_0(z), \pi_1(z)\rangle \mid z \in s\} \cup \{\langle l, \langle k, 0, 0\rangle\rangle\} = \\
\quad \{\langle 1, \langle 1, m, n\rangle\rangle\} \cup \{\langle \pi_0(z) + 1, \pi_2(z)\rangle \mid z \in s\}
\end{array}
$$

The statement is now straightforward to prove through an argument similar to *shifted pairing* [17, Theorem 15]. ⊠

The following is an immediate consequence of the proof of Theorem 2.

Corollary 1. *$TS(\mathcal{P})$ is undecidable. Undecidability arises already for formulas of the form $\exists v \exists x \varphi$, where φ is quantifier free and uses at most three unnested comprehensions.*

The construction of halts_M in Theorem 2 shows that comprehensions together with pairing (or tuples) leads to undecidability of the one step reachability problem, because valid_M can be used as an enabling condition of an action as illustrated in Figure 2, and the halting problem of 2-register machines is undecidable.

Only a small fragment of Presburger arithmetic is needed. In particular, divisibility by a constant is not needed. The proof of the theorem does not change if M is assumed to be a Turing machine (assume M has two input symbols and the configuration (i, m, n) represents a snapshot of M where i is the finite state of M, m represents the tape content to the left of the tape head and n represents the tape content to the right of the tape head), only the construction of `STEP` is

different. However, in that case one needs to express divisibility by 2 to determine the input symbol represented by the lowest bit of the binary representation of m or n, which can be encoded using an additional existential quantifier.

Another consequence of the construction in Theorem 2 is that decidability of the bounded reachability problem cannot in general be obtained by fixing the model program or by limiting the number of set variables (without disallowing them).

Corollary 2. *There is a fixed model program P_u over $TS(\mathcal{P})$ with one set-valued state variable, one integer-valued state variable, and an action symbol with two integer-valued parameters, such that the following problem is undecidable: given an action a, decide if a is enabled in P_u.*

Proof. Let M_u be a 2-register machine that is *universal* in the following sense, given a Turing machine M and an input v (over a fixed alphabet), let $\ulcorner M, v \urcorner$ be an effective encoding of M and v as an input for M_u, so that M_u accepts $\ulcorner M, v \urcorner$ if and only if M accepts v. Such a 2-register machine exists and can be constructed effectively [21, Theorem 7.9]. Let P_u be like P_{M_u} in Figure 2. Let M be a Turing machine and v an input for M. Then $\texttt{halts}_{M_u}(\ulcorner M, v \urcorner)$ is enabled in P_u iff (by Theorem 2) M_u halts on $\ulcorner M, v \urcorner$ iff M accepts v. ⊠

A *basic* value or sort is a non-set value or sort. A parameter or state variable is *basic* if its sort is basic.

Definition 4. A model program is *basic* if all of its action parameters are basic, each state variable is either basic or a set of basic elements, and set-sorted state variables are initialized with expressions that contain no set-sorted state variables.

Example 3. The model program P_u in Corollary 2 is not basic because the initial value of `steps` is undefined. The following model program on the other hand is basic, where `STEP` and `k` are the same as above.

```
[Action] halts(maxCounter as Integer, l as Integer)
  let steps = {(j,(i,m,n),(i',m',n')) | i,i' in {1..k}, j in {1..l},
               m,n,m',n' in {1..maxCounter}, STEP((i,m,n),(i',m',n'))}
  require {(j,x) | (j,x,y) in steps} union {(1,(k,0,0))} =
          {(1,(1,m,n))} union {(j+1,y) | (j,x,y) in steps}
```

It seems as if it is possible to express the halting problem just using bounded reachability of basic model programs. This is not the case as is shown in Section 4. Intuitively, a comprehension adds "too many" elements.

An extension of basic model programs that leads to undecidability of the one step reachability problem is if we allow *set cardinality*. We can then express integer multiplication as follows, given two (non-negative) integers m and n: $m \cdot n \stackrel{\text{def}}{=} |\{1..m\} \times \{1..n\}|$. Also, if we allow *bag comprehensions* we can define the cardinality of a set s as $|s| \stackrel{\text{def}}{=} \{\!\{0 | x \in s\}\!\}[0]$. Either of these extensions allows us to effectively encode diophantine equations (e.g. $5x^2y + 6z^3 - 7 = 0$ is a diophantine equation). Let $p(\overline{x})$ be a diophantine equation and let $\texttt{P}(\overline{x})$ be an action whose enabling condition is the encoding of $p(\overline{x})$. Then $\texttt{P}(\overline{n})$ is enabled iff $\overline{n}$ is an integer solution for $p(\overline{x})$. The problem of deciding whether a diophantine equation has an integer solution is known as *Hilbert's 10th problem* and is undecidable [28].

4 Bounded Reachability of Basic Model Programs

We show that the bounded reachability problem of *basic* model programs over a background $TS(\mathcal{A})$ is decidable provided that $Th(\mathcal{A})$ is decidable, where $Th(\mathcal{A})$ is the closure of $\mathcal{A}$ under entailment. We say that $Th(\mathcal{A})$ is decidable, if for an arbitrary closed first-order formula φ in the language of $\mathcal{A}$ it is decidable whether $\varphi \in Th(\mathcal{A})$.

The proof has two steps. First, we show that there is a fragment of $TS(\mathcal{A})$ formulas, denoted by $TS(\mathcal{A})_{\prec}$, for which the validity or satisfiability problem reduces effectively to $\mathcal{A}$, by showing that there is an effective equivalence preserving mapping from formulas in $TS(\mathcal{A})_{\prec}$ to formulas in $\mathcal{A}$. Second, we show that the bounded reachability problem of basic model programs over $TS(\mathcal{P})$ reduces to (satisfiability in) $TS(\mathcal{P})_{\prec}$. Let $\mathcal{A}$ be fixed. Let $V(\varphi)$ denote the collection of all set variables that occur in a formula φ over $TS(\mathcal{A})$.

Definition 5. A $TS(\mathcal{A})$ formula φ is in $TS(\mathcal{A})_{\prec}$ (also called *stratified*) if

- φ has the form $\psi \wedge \bigwedge_{v \in V(\varphi)} v = S_v$, and
- the relation $\prec \stackrel{\text{def}}{=} \{(w, v) | v \in V(\varphi), w \in V(S_v)\}$ is well-founded.

The equation $v = S_v$ is called the *definition of* v in φ.

Theorem 3. *$TS(\mathcal{A})_{\prec}$ reduces effectively to $\mathcal{A}$.*

Proof (Outline). The definition of $TS(\mathcal{A})_{\prec}$ is equivalent to the following construction in the case when all tuples are required to be flat. Let L_0 be the language of $\mathcal{A}$ and let $\mathcal{A}_0 = \mathcal{A}$. Given L_i and $\mathcal{A}_i$, create L_{i+1} and $\mathcal{A}_{i+1}$ as follows: expand L_i with a relation symbol R_φ of arity n for each L_i-formula $\varphi(x_1, \ldots, x_n)$ and add the definition $\forall \overline{x}(R_\varphi(\overline{x}) \leftrightarrow \varphi(\overline{x}))$ to $\mathcal{A}_i$. Now $TS(\mathcal{A})_{\prec}$ corresponds to $\bigcup_i \mathcal{A}_i$ as follows. Due to the well-founded ordering, each set variable v with the definition $v = \{\langle \overline{x} \rangle \mid_{\overline{x}} \varphi(\overline{x})\}$ corresponds to a relation symbol R_φ. Given a formula φ in $TS(\mathcal{A})_{\prec}$, it corresponds thus to a formula φ_k in $\mathcal{A}_k$ for some k. The statement follows by using the theorem of the existence of definitional expansions [20, Theorem 2.6.4] to reduce φ_{i+1} in L_{i+1} to an equivalent φ_i in L_i. ⊠

It follows that $TS(\mathcal{P})_{\prec}$ is decidable. We also get the following corollary that is the main result of this section.

Corollary 3. *Bounded reachability of basic model programs over $TS(\mathcal{P})$ is decidable.*

Proof. Let P be a basic model program over $TS(\mathcal{P})$ let φ be a reachability condition, and let k be a step bound. It is easy to see that $\psi = Reach(P, \varphi, k)$

can be written as a stratified $TS(\mathcal{P})$ formula: First, we can assume that there is only one action symbol (with a specific parameter that identifies a particular action). Since P is basic, the initial value of each state variable $v_{(0)}$ must be defined. In each step formula for step i, the value $v_{(i+1)}$ is given a definition that uses only variables or parameters from state i and parameters are basic. The definition can be written on a form that uses *ite* and is a top level equation of the generated formula. The only variables that are not given definitions are parameters, but all parameters are basic. Satisfiability of ψ in the language that includes the state variables reduces to entailment of the existential closure of ψ from $TS(\mathcal{P})$, which by Theorem 3, reduces to $\mathcal{P}$ and is thus decidable. ⊠

General integer arrays and array read and write operations are, strictly speaking, not in the $TS(\mathcal{P})$ fragment but can easily be encoded using tuples and comprehensions. For example, given an array variable v from integers to integers with the default value 0, encode it as the graph $\tilde{v}$ of v. The relation $Read(\tilde{v}, l, x)$ that holds when $v[l] = x$, can be defined through $Read(\tilde{v}, l, x) \stackrel{\text{def}}{=} ite(\{x\} = \{\pi_1(y) \mid y \in \tilde{v} \wedge \pi_0(y) = l\}, true, x = 0)$ and the corresponding write operation $Write(\tilde{v}, l, x)$ can be defined through $Write(\tilde{v}, l, x) \stackrel{\text{def}}{=} \{y \mid y \in \tilde{v} \wedge \pi_0(y) \neq l\} \cup \{(l, x)\}$. Using this encoding one can for example transform the *Credits* model program in Example 1 into an equivalent model program over $TS(\mathcal{P})$.

5 Implementation

We use the state of the art Z3 SMT solver for the implementation of bounded model checking. The initial implementation was described in [36]. We have extended this work in several aspects, including support for comprehensions with multiple comprehension variables and non-invertible comprehension expressions, bag(multiset) support etc., all of which make use of the iterated model refinement technique (explained in the following paragraphs). This is possible due to the fact that model programs are executable, so the feasibility of traces provided by the solver can be checked.

While, in principle, the traces could be executed directly on the model program via the ASML compiler, we use the approach to translate them to C# and executed the traces on C# code. This provides several benefits: we can conveniently use .Net API's for reflection, we can add auxiliary methods for evaluating and saving intermediate results for pinpointing error locations (in case an erroneous trace is provided by the solver) etc. Additionally, this eases the adoption of other languages for describing model programs. For example, NModel [30] uses C# as the modelling language. In this case, we would only need to provide a parser from C# to the internal abstract syntax to be able to use the framework.

The refinement loop works as follows. A trace provided by the Z3 solver is executed step by step on the generated program via reflection, and after each step it is checked whether the state given in the model matches the actual state (simply by comparing the variable values as assigned by the solver to the values in the actual state). If it does not match, we know at which action the mismatching

state was reached. By examining the statements in the action we can check which of the axioms was not instantiated correctly and on which variables, consequently pinpointing the exact error source. The interpretation on this operation can then be fixed, by adding new formulas to the original model formula, giving explicit instantiations of the "misinterpreted" axiom on each index term. The new formula can be sent back to Z3 and a new trace obtained, which might again be erroneous (on some other axiom application), in which case it is again fixed and the refinement loop continues. This technique helps us circumvent SMT solvers' difficulties in coping with quantifiers. The approach is similar to CEGAR [9] (counter example guided abstraction refinement), the main difference being that we do not refine the level of abstraction, but instead lazily instantiate axioms in case their use has not been triggered during proof search.

We make use of the iterative refinement in several cases. In comparison to [36], we have added support for comprehensions which include more than 1 variable, and for the case where the element term is not invertible. In this case we rewrite the comprehension into formulas $\forall \overline{y}(\varphi[\overline{x}] \rightarrow t[\overline{x}] \in s')$ and $\forall y(y \in s' \rightarrow \exists \overline{x}(y = t[\overline{x}] \wedge \wedge \varphi[\overline{x}]))$. Existential quantifiers can be eliminated from the latter by skolemization. During the refinement loop, the two axioms are instantiated for specific index terms if needed.

Similar approach is used when extending the framework with bag support. While adding the bag axioms to Z3 is straightforward (for example the definition of bag union is $\forall x, s_1, s_2.((s_1 \uplus s_2)[x] \equiv s_1[x] + s_2[x])$), traces given by Z3 might be incorrect. (Quantifiers are implemented via pattern matching in Z3, so axioms are instantiated only if the particular pattern is encountered during proof search. If the pattern is not encountered, the axiom never gets used.) In this case, we can again use the model checking technique to pinpoint the source of error, and refine the model. This procedure is complete in case of integer bags.

6 Related Work

The *unbounded* reachability problem for model programs without comprehensions and with parameterless actions is shown to be undecidable in [13], where it is called the hyperstate reachability problem. General reachability problems for transition systems are discussed in [33] where the main results are related to guarded assignment systems. A guarded assignment system is a union of guarded assignments or update rules. Detailed proofs of the theorems in this paper are given in [38]. The case when $\mathcal{A} = \mathcal{P}$ in Theorem 3 is related to decidable extensions of $\mathcal{P}$ that are discussed in [4].

The decidable fragment BAPA [26] is an extension of Boolean algebra with $\mathcal{P}$. The sets in BAPA are finite and bounded by a maximum size and the cardinality operator is allowed, which unlike for $TS(\mathcal{P})_{\prec}$, does not enable encoding of multiplication. Comprehensions are not possible and the element-of relation is not allowed, i.e. integers and sets can only be related through the cardinality operator. A decidable fragment of bag (multiset) constraints combined with summation constraints are considered in [31] where summation constraints can be used to express set cardinality (without using bag cardinality that is also

included in the fragment). A related fragment of integer linear arithmetic with a star operator is considered in [32].

In [8] a decision procedure for an array fragment is introduced and in [36] it is shown that this decision procedure can be applied to the bounded reachability problem of a subclass of model programs. However, the fragment in [8] does not allow expressions that include universally quantified variables, other than the variable itself, to occur in array read operations. Consequently, comprehensions where the comprehension expression is not invertible are not covered in [36]. In [19] another fragment of arrays is considered that allows universal variables in array read expressions that relate consecutive elements or talk about periodic properties.

The full fragment $TS(\mathcal{P})$ is also part of the data structures that are allowed in the Jahob verification system [7]. Formulas in this fragment are translated in Jahob to standard first-order formulas that can be proven using a resolution theorem prover.

A technique for translating common comprehension expressions (such as *sum* and *count*) into verification conditions is presented in [27] within the Spec# verification system that uses Boogie to generate verification conditions for SMT solvers [3]. The system does not support arbitrary set comprehension expressions as terms but allows axioms that enable explicit definitions of sets.

The reduction of the theories of arrays, sets and multisets to the theory of equality with uninterpreted function symbols and linear arithmetic is used in [24] for constructing interpolants for these theories. This work is based on the results of [25], where it is shown that the quantifier-free theories of arrays, sets and multisets can be reduced to quantifier-free theories of uninterpreted symbols with equality, constructors and Presburger arithmetic.

Using SAT for bounded reachability of transition systems was introduced in in [5] and the extension to SMT was introduced in [1]. Besides Z3 [10], other SMT solvers that support arrays are described in [2,35]. The formula encoding we use [36] into SMT follows the same scheme but does not unwind comprehensions and makes the action label explicit.

Our quantifier elimination scheme is inspired by [8], and refines it by using model-checking to implement an efficient incremental saturation procedure on top of the SMT solver. The work here extends the work in [36] through support for set comprehensions with multiple comprehension variables and non-invertible comprehension expressions, as well as bag (multi-set) axioms. A recent application of the quantifier elimination scheme has been pursued by [23] in the context of railway control systems.

The following problems have not been addressed yet. Bounded reachability of model programs that use nested comprehensions, including for example sets and bags, is interesting for analysis of general purpose algorithms, see e.g. [18]. Given the (computational) complexity of $\mathcal{A}$, what is the complexity of $TS(\mathcal{A})_{\prec}$? It seems that a $TS(\mathcal{A})_{\prec}$ formula can be exponentially more succinct than the corresponding $\mathcal{A}$ formula. So, the complexity of $TS(\mathcal{P})_{\prec}$ could thus be $2^{2^{2^{cn}}}$, since the complexity of $\mathcal{P}$ is $2^{2^{cn}}$ [12]. The proper instantiation of array indices and avoidance of false models generated by an SMT solver, due to the inherent

incompleteness of the triggering mechanism of universally quantified axioms, is an important open problem in the general case.

Acknowledgement. We thank Nikolaj Bjørner for support with Z3. We also thank the anonymous referees for their helpful comments and suggestions. The second author received support from the Estonian Doctoral School in ICT, the EITSA Tiger University Plus programme and the Estonian Association of Information Technology and Telecommunications (ITL).

References

1. Armando, A., Mantovani, J., Platania, L.: Bounded model checking of software using SMT solvers instead of SAT solvers. In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 146–162. Springer, Heidelberg (2006)
2. Armando, A., Ranise, S., Rusinowitch, M.: A rewriting approach to satisfiability procedures. Inf. Comput. 183(2), 140–164 (2003)
3. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
4. Bès, A.: A survey of arithmetical definability, A tribute to Maurice Boffa, Special Issue of Belg. Math. Soc., 1–54 (2002)
5. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
6. Börger, E., Grädel, E., Gurevich, Y.: The Classical Decision Problem. Springer, Heidelberg (1997)
7. Bouillaguet, C., Kuncak, V., Wies, T., Zee, K., Rinard, M.: On using first-order theorem provers in the Jahob data structure verification system. Technical Report MIT-CSAIL-TR-2006-072, Massachusetts Institute of Technology (November 2006)
8. Bradley, A.R., Manna, Z., Sipma, H.B.: What's decidable about arrays? In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 427–442. Springer, Heidelberg (2006)
9. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
10. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963. Springer, Heidelberg (2008)
11. de Moura, L.M., Rueß, H., Sorea, M.: Lazy theorem proving for bounded model checking over infinite domains. In: Voronkov, A. (ed.) CADE 2002. LNCS, vol. 2392, pp. 438–455. Springer, Heidelberg (2002)
12. Fischer, M.J., Rabin, M.O.: Super-exponential complexity of Presburger arithmetic. In: SIAMAMS, pp. 27–41 (1974)
13. Grieskamp, W., Gurevich, Y., Schulte, W., Veanes, M.: Generating finite state machines from abstract state machines. SIGSOFT Softw. Eng. Notes 27(4), 112–122 (2002)
14. Grieskamp, W., MacDonald, D., Kicillof, N., Nandan, A., Stobie, K., Wurden, F.: Model-based quality assurance of Windows protocol documentation. In: ICST 2008, Lillehammer, Norway (April 2008)

15. Gurevich, Y.: Evolving Algebras 1993: Lipari Guide. In: Specification and Validation Methods, pp. 9–36. Oxford University Press, Oxford (1995)
16. Gurevich, Y., Rossman, B., Schulte, W.: Semantic essence of AsmL. Theor. Comput. Sci. 343(3), 370–412 (2005)
17. Gurevich, Y., Veanes, M.: Logic with equality: partisan corroboration and shifted pairing. Inf. Comput. 152(2), 205–235 (1999)
18. Gurevich, Y., Veanes, M., Wallace, C.: Can abstract state machines be useful in language theory? Theor. Comput. Sci. 376(1), 17–29 (2007)
19. Habermehl, P., Iosif, R., Vojnar, T.: What else is decidable about arrays? In: Amadio, R. (ed.) FOSSACS 2008. LNCS, vol. 4962. Springer, Heidelberg (2008)
20. Hodges, W.: Model theory. Cambridge Univ. Press, Cambridge (1995)
21. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Reading (1979)
22. Jacky, J., Veanes, M., Campbell, C., Schulte, W.: Model-based Software Testing and Analysis with C#. Cambridge University Press, Cambridge (2008)
23. Jacobs, S., Sofronie-Stokkermans, V.: Applications of hierarchical reasoning in the verification of complex systems. ENTCS 174(8), 39–54 (2007)
24. Kapur, D., Majumdar, R., Zarba, C.G.: Interpolation for data structures. In: SIGSOFT FSE 2006, pp. 105–116. ACM, New York (2006)
25. Kapur, D., Zarba, C.G.: A reduction approach to decision procedures (2006)
26. Kuncak, V., Nguyen, H.H., Rinard, M.: An algorithm for deciding BAPA: Boolean algebra with Presburger arithmetic. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS, vol. 3632, pp. 260–277. Springer, Heidelberg (2005)
27. Leino, R., Monahan, R.: Automatic verification of textbook programs that use comprehensions. In: FTfJP 2007, Berlin, Germany (July 2007)
28. Matiyasevich, Y.V.: Hilbert's tenth problem. MIT Press, Cambridge (1993)
29. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Trans. Program. Lang. Syst. 1(2), 245–257 (1979)
30. NModel. Public version released (May 2008), http://www.codeplex.com/NModel
31. Piskac, R., Kuncak, V.: Decision procedures for multisets with cardinality constraints. In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) VMCAI 2008. LNCS, vol. 4905, pp. 218–232. Springer, Heidelberg (2008)
32. Piskac, R., Kuncak, V.: On Linear Arithmetic with Stars. Technical Report LARA-REPORT-2008-005, EPFL (2008)
33. Rybina, T., Voronkov, A.: A logical reconstruction of reachability. In: Broy, M., Zamulin, A.V. (eds.) PSI 2003. LNCS, vol. 2890, pp. 222–237. Springer, Heidelberg (2004)
34. SMB2 (2008), http://msdn2.microsoft.com/en-us/library/cc246482.aspx
35. Stump, A., Barrett, C.W., Dill, D.L., Levitt, J.R.: A decision procedure for an extensional theory of arrays. In: LICS 2001, pp. 29–37. IEEE, Los Alamitos (2001)
36. Veanes, M., Bjørner, N., Raschke, A.: An SMT approach to bounded reachability analysis of model programs. In: Suzuki, K., Higashino, T., Yasumoto, K., El-Fakih, K. (eds.) FORTE 2008. LNCS, vol. 5048. Springer, Heidelberg (2008)
37. Veanes, M., Campbell, C., Grieskamp, W., Schulte, W., Tillmann, N., Nachmanson, L.: Model-based testing of object-oriented reactive systems with Spec Explorer. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) FORTEST 2008. LNCS, vol. 4949, pp. 39–76. Springer, Heidelberg (2008)
38. Veanes, M., Saabas, A., Bjørner, N.: Bounded reachability of model programs. Technical Report MSR-TR-2008-81, Microsoft Research (May 2008)

Program Complexity in Hierarchical Module Checking*

Aniello Murano[1], Margherita Napoli[2], and Mimmo Parente[2]

[1] Università di Napoli "Federico II", Via Cintia, 80126 - Napoli, Italy
[2] Università di Salerno, Via Ponte don Melillo, 84084 - Fisciano (SA), Italy

Abstract. *Module checking* is a well investigated technique for verifying the correctness of open systems, which are systems characterized by an ongoing interaction with an external environment. In the classical module checking framework, in order to check whether an open system satisfies a required property, we first translate the entire system into an open model (*module*) that collects all possible behaviors of the environment and then check it with respect to a formal specification of the property.

Recently, in the case of closed system, Alur and Yannakakis have considered hierarchical structure models in order to have models exponentially more succinct. A hierarchical model uses as nodes both ordinary nodes and supernodes, which are hierarchical models themselves. For *CTL* specifications, it has been shown that for the simple case of models having only single-exit supernodes, the hierarchical model checking problem is not harder than the classical one. On the contrary, for the more general multiple-exit case, the problem becomes PSPACE-complete.

In this paper, we investigate the program complexity of the CTL *hierarchical module checking problem*, that is, we consider the module checking problem for a fixed *CTL* formula and modules having also supernodes that are modules themselves. By exploiting an automata-theoretic approach through the introduction of hierarchical Büchi tree automata, we show that, in the single-exit case, the addressed problem remains in PTIME, while in the multiple-exit case, it becomes PSPACE-complete.

1 Introduction

Module checking is a useful technique that allows to verify the correctness of open systems [KVW01]. While the behavior of a closed system is fully characterized by internal states, an open system maintains an ongoing interaction with an external environment, and its behavior is fully affected by this interaction.

Classically, in order to check whether an open system satisfies a required property, we translate the entire system into a *module*, that is in a labeled state-transition graph whose set of states is partitioned into a set of system states (where the system makes a transition) and a set of environment states (where

* Work partially supported by MIUR PRIN Project no.2007-9E5KM8 and grant "Formal Methods for Closed and Open Systems" ex-60% 2006 Università di Salerno.

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 318–332, 2008.

the environment makes a transition). Given a module $\mathcal{M}$, describing the system to be verified, and a temporal logic formula φ, specifying the desired behavior of the system, module checking asks whether, for all possible environments, $\mathcal{M}$ satisfies φ. Therefore, while in *model checking* it is sufficient to check whether the full computation tree obtained by unwinding $\mathcal{M}$ satisfies φ, in module checking it is necessary to verify that all trees obtained from the full computation tree by pruning some subtrees rooted in nodes corresponding to choices disabled by the environment (those trees represent the interactions of $\mathcal{M}$ with all the possible environments) satisfy φ. We collect all such trees in a set named $exec(\mathcal{M})$.

As a classic example of closed and open systems, we can think of two drink-dispensing machines. One machine, which is a closed system, repeatedly boils water, makes an internal nondeterministic choice, and serves either coffee or tea. The second machine, which is an open system, repeatedly boils water, asks the environment to choose between coffee and tea, and deterministically serves a drink according to the external choice. Both machines induce the same infinite tree of possible executions. Nevertheless, while the behavior of the first machine is determined by internal choices solely, the behavior of the second machine is determined also by external choices, made by its environment. Formally, in a closed system, the environment cannot modify any of the system variables. In contrast, in an open system, the environment can modify some of them. In [KVW01, AMV07], it has been shown that for systems modeled as single modules and specifications as branching time temporal logic formulas, module checking is exponentially harder than model checking.

In formal verification a very interesting question is how complex is to check for system correctness in the case we fix the specification. This question is usually addressed as the *program complexity* and in more details concerns the complexity of the verification question for the set $\{\mathcal{M} \mid \mathcal{M} \text{ satisfies } \varphi\}$, for a fixed formula φ[VW86]. Program complexity is receiving great attention in formal verification due to the fact that often the size of the system widely exceeds that of the formula, which is usually very small and therefore considered constant. This allows us to use in practice formal verification techniques whenever they result tractable with respect to the system. For example, we recall that for finite-state systems and specifications given as formulas of the branching-time temporal logic *CTL* ([CE81]), module checking is EXPTIME-complete, but the corresponding problem with a constant size formula is only PTIME-complete [KVW01].

Recently, in the case of complex closed systems, hierarchical structure models have been usefully considered, in order to have models exponentially more succinct. A hierarchical model uses as nodes both ordinary nodes or supernodes, which are models themselves [AY01, ABE+05, LNPP08]. The straightforward way to analyze a hierarchical closed machine is to flatten it (thus, incurring an exponential blow up) and apply a model checking tool on the resulting ordinary model. In [AY01], it has been shown that for linear-time specifications such as *LTL*, the cost of flattening can be avoided by showing that the *LTL* hierarchical model checking problem is not harder than the classical one. The same happens for *CTL* specifications, for models having one exit node. In the general case,

instead, *CTL* hierarchical model checking becomes exponentially harder and, in particular, the program complexity is PSPACE-complete.

In this paper, we investigate the program complexity of the module checking problem for *CTL* in the case of modules expressed by *hierarchical modules*, that is nodes of the module can be ordinary nodes or supernodes which are modules themselves. As a simple example, consider again the above drink-dispensing machine. Now suppose that both in the cases the environment makes a coffee or tea choice, the system allows the environment to have an extra choice between regular sugar or diet sugar. In both cases, we can remand the choice to another module. As for non-hierarchical open systems, in case we want to check whether it is possible for the designed hierarchical open machine to serve coffee with regular sugar, a straightforward way is to flatten it and then, by using the classical module checking technique, check whether the flatten module satisfies the *CTL* formula $AGEF$ *coffe_with_regular_sugar*. Unfortunately, by flattening the hierarchical module, we increase exponentially the size of the module and we immediately get that *CTL* hierarchical module checking is EXPTIME w.r.t. both the sizes of the hierarchical module and the formula.

In this paper, we show that for the addressed problem the cost of flattening can be avoided. In particular, we show that for single-exit hierarchical modules, the program complexity of the *CTL* hierarchical module checking problem is not harder than the classical one, while in the case of multiple-exit hierarchical modules, the addressed problem becomes PSPACE-complete.

For the upper bounds, we use an automata-theoretic approach via tree automata. In particular, we introduce *hierarchical nondeterministic Büchi tree automata* (*HNBT*) and use a reduction to the emptiness problem for this automata. In more details, given a hierarchical module $\mathcal{M}$ and a *CTL* formula φ, we first construct in polynomial time an *HNBT* $\mathcal{A}_{\mathcal{M}}$ accepting $exec(\mathcal{M})$. The construction of $\mathcal{A}_{\mathcal{M}}$ we propose here extends that used in [KVW01] by also taking into account that $\mathcal{M}$ is in a hierarchical shape. Thus, $\mathcal{A}_{\mathcal{M}}$ will have, for each supernode in the hierarchical module (which is a hierarchical module itself) a corresponding supernode (which is a hierarchical automaton itself) with the same number of exit nodes. From the formula side, accordingly to [KVW00], we construct in exponential time a nondeterministic Büchi tree automaton (*NBT*) $\mathcal{A}_{\neg\varphi}$ accepting all models that do not satisfy φ, with the intent to check that none of them are in $exec(\mathcal{M})$. Thus, we check that $\mathcal{M}$ models φ for every possible choice of the environment by checking whether $\mathcal{L}(\mathcal{A}_{\mathcal{M}}) \cap \mathcal{L}(\mathcal{A}_{\neg\varphi})$ is empty. To obtain the result, we first show that the product of the *HNBT* $\mathcal{A}_{\mathcal{M}}$ with the *NBT* $\mathcal{A}_{\neg\varphi}$ can be performed in polynomial time, turning into an *HNBT* having a number of exit nodes that depends on the number of states of $\mathcal{A}_{\neg\varphi}$, which in turn depends on the size of φ. Since we are interested on the program complexity of the hierarchical module checking problem, we assume the formula to be fixed. Therefore, the obtained *HNBT* will have multiple exits if $\mathcal{A}_{\mathcal{M}}$ does, and a constant number of exit-nodes otherwise. Then, we show that the emptiness problem for an *HNBT* can be solved in PTIME if it only admits constant (and in particular single-) exits, while it is PSPACE-complete in the case of multiple exits. Thus, we get the desired upper bounds. To show

matching lower bounds, for the single-exit case, we recall that the program complexity of the classical module checking problem is PTIME-hard and that, for multiple exits, the program complexity of *CTL* hierarchical model checking closed system is PSPACE-hard.

The paper is self-contained and is organized as follows. In the next section, we give basic definitions, introduce open hierarchical state machines, and define the hierarchical module checking problem for *CTL*. In Section 3, we briefly recall *NBT* and introduce *HNBT*. Then, we solve the emptiness problem for *HNBT*. Finally, in Section 4, we solve the hierarchical module checking problem for *CTL*.

2 Preliminary

In this section, we introduce the *hierarchical module checking problem* for *CTL*.

Let $\mathbb{N}$ be the set of positive integers. A *tree* T is a prefix closed subset of $\mathbb{N}^*$. The elements of T are called *nodes* and the empty word ε is the *root* of T. For $x \in T$, the set of *children* of x (in T) is $children(T, x) = \{x \cdot i \in T \mid i \in \mathbb{N}\}$. For $k \geq 1$, the complete k-ary tree is the tree $\{1, \ldots, k\}^*$. For $x \in T$, a path π of T from x is a set $\pi \subseteq T$ such that $x \in \pi$ and for each $y \in \pi$ such that $children(T, y) \neq \emptyset$, there is exactly one node in $children(T, y)$ belonging to π. In the following, for *a path of* T, we mean a path of T from the root ε. For an alphabet Σ, a Σ-labeled tree is a pair (T, V), where T is a tree and $V : T \to \Sigma$ maps each node of T to a symbol in Σ.

In this paper, we consider open systems, i.e. systems that interact with their environment and whose behavior depends on this interaction. The global behavior of such a system is described by a finite state machine (also called *module* [KVW01]) $\mathcal{M} = (AP, S, E, R, in, L)$, where AP is a finite set of atomic propositions, $S \cup E$ is a finite set of states partitioned into a set S of *system* states and a set E of *environment* states (we use W to denote $S \cup E$), $R \subseteq W \times W$ is a total transition relation, $in \in W$ is an initial state, and $L : W \to 2^{AP}$ maps each state w to the set of atomic propositions that hold in w. For $(w, w') \in R$, we say that w' is a successor of w. For each state $w \in W$, we denote by $succ(w)$ the ordered tuple of w's successors. When the module $\mathcal{M}$ is in a system state w_s, then all the states in $succ(w_s)$ are possible next states. On the other hand, when $\mathcal{M}$ is in an environment state w_e, then the possible next states (that are in $succ(w_e)$) depend on the current environment. Since the behavior of the environment is not predictable, we have to consider all the possible sub-tuples of $succ(w_e)$. The only constraint, since we consider environments that cannot block the system, is that at least one transition from w_e exists leading into a next state in $succ$ (not all these transitions may be disabled by the environment).

The set of all the maximal computations of $\mathcal{M}$ starting from the initial state in is described by a W-labeled tree $(T_{\mathcal{M}}, V_{\mathcal{M}})$, called *computation tree*, which is obtained by unwinding $\mathcal{M}$ in the usual way. The problem of deciding, for a given branching-time formula ψ over AP, whether $(T_{\mathcal{M}}, L \circ V_{\mathcal{M}})$ satisfies ψ, denoted $\mathcal{M} \models \psi$, is the usual *model-checking problem* [CE81, QS81]. On the other hand, for an open system, $(T_{\mathcal{M}}, V_{\mathcal{M}})$ corresponds to a very specific environment, i.e. a

maximal environment that never restricts the set of its next states. Therefore, when we examine a branching-time specification ψ w.r.t. a module $\mathcal{M}$, ψ should hold not only in $(T_{\mathcal{M}}, V_{\mathcal{M}})$, but also in all the trees obtained by pruning from $(T_{\mathcal{M}}, V_{\mathcal{M}})$ subtrees whose root is a child (successor) of a node corresponding to an environment state. The set of these labeled trees is denoted by $exec(\mathcal{M})$, and is formally defined as follows. $(T, V) \in exec(\mathcal{M})$ iff $T \subseteq T_{\mathcal{M}}$, V is the restriction of $V_{\mathcal{M}}$ to the tree T, and for all $x \in T$ the following holds:

- if $V_{\mathcal{M}}(x) = w \in S$ and $succ(w) = \langle w_1, \ldots, w_n \rangle$, then $children(T, x) = \{x \cdot 1, \ldots, x \cdot n\}$ (note that for $1 \leq i \leq n$, $V(x \cdot i) = V_{\mathcal{M}}(x \cdot i) = w_i$);
- if $V_{\mathcal{M}}(x) = w \in E$ and $succ(w) = \langle w_1, \ldots, w_n \rangle$, then there is a subtuple $\langle w_{i_1}, \ldots, w_{i_p} \rangle$ of $succ(w)$, with $p \geq 1$, such that $children(T, x) = \{x \cdot i_1, \ldots, x \cdot i_p\}$ (note that for $1 \leq j \leq p$, $V(x \cdot i_j) = V_{\mathcal{M}}(x \cdot i_j) = w_{i_j}$).

Intuitively, each labeled tree (T, V) in $exec(\mathcal{M})$ corresponds to a different behavior of the environment. In the following, we consider the trees in $exec(\mathcal{M})$ as 2^{AP}-labeled trees, i.e. taking the label of a node x to be $L(V(x))$.

In this paper, we consider the branching-time temporal logic *CTL* as system specification. *CTL* was introduced by Emerson and Clarke in 1981 [CE81] as a tool for specifying and verifying concurrent programs. *CTL* formulas are built from a set AP of *atomic propositions* using boolean operators, the linear-temporal operators X ("next time") and U ("until"), coupled with the path quantifiers A ("for all paths") or E ("for some path"). For a formal definition of *CTL* see [CGP99]. The *closure* $cl(\varphi)$ of a *CTL* formula φ is the set of all subformulas of φ, including φ. The size $|\varphi|$ of φ is defined as the number of elements in $cl(\varphi)$. Given a *CTL* formula φ, we say that (T, V) satisfies φ if $((T, V), \varepsilon) \models \varphi$.

For a module $\mathcal{M}$ and a *CTL* formula ψ, we say that $\mathcal{M}$ satisfies ψ, denoted $\mathcal{M} \models_r \psi$, if all the trees in $exec(M)$ satisfy ψ. The problem of deciding whether $\mathcal{M}$ satisfies ψ is called *module checking* [KVW01]. Note that $\mathcal{M} \models_r \psi$ implies $\mathcal{M} \models \psi$ (since $(T_{\mathcal{M}}, V_{\mathcal{M}}) \in exec(\mathcal{M})$), but the converse in general does not hold. Also, note that $\mathcal{M} \not\models_r \psi$ is *not* equivalent to $\mathcal{M} \models_r \neg\psi$. Indeed, $\mathcal{M} \not\models_r \psi$ just states that there is some tree $(T, V) \in exec(\mathcal{M})$ satisfying $\neg\psi$.

Open Hierarchical State Machines. An *open hierarchical state machine*, or *hierarchical module* $\mathcal{M}$ over a set AP of atomic propositions is a tuple $(\mathcal{M}_1, \ldots, \mathcal{M}_n)$ of *components*, where each $\mathcal{M}_i = (AP, S_i, E_i, R_i, Box_i, O_i, in_i, L_i, Y_i)$, $1 \leq i \leq n$, has the following elements:

- A finite set S_i of *system nodes*;
- A finite set E_i of *environment nodes*. We assume $S_i \cap E_i = \emptyset$, and $W_i = S_i \cup E_i$;
- A finite set Box of *boxes* (or supernodes). We assume $W_i \cap Box_i = \emptyset$;
- An initial node in_i of W_i;
- A subset O_i of W_i, called exit-nodes.
- A labeling function $L_i : W_i \to 2^{AP}$ labeling each node with a subset of AP.
- An indexing function $Y_i : Box_i \to \{i + 1, \ldots, n\}$ that maps each box of the i-th component to an index greater than i. That is, if $Y_i(b) = j$, for a box b of $\mathcal{M}_i$, then b can be viewed as a reference to the component $\mathcal{M}_j$.

– An edge relation R_i. Each edge in R_i is a pair (u, v) with source u and sink v: source u either is a node of $\mathcal{M}_i$, or is a pair (u_1, u_2), where u_1 is a box of $\mathcal{M}_i$ with $Y_i(u_1) = j$ and u_2 is an exit-node of $\mathcal{M}_j$, and the sink v is either a node or a box of $\mathcal{M}_i$.

The edges connect nodes and boxes with one another. Edges entering a box implicitly connect to the unique initial node of the component associated with that box. On the other hand, edges exiting a box explicitly specify an exit-node among the possible exit-nodes of the component associated with that box. A hierarchical module is closed (called *hierarchical model* in [AY01]) if for all components $\mathcal{M}_i$, we have $E_i = \emptyset$.

By extending an idea used for closed hierarchical models, we can associate to a hierarchical module an ordinary flat module, by recursively substituting each box with the component indexed by the box. Since different boxes can be associated with the same component, each node can appear in different contexts. The expanded flat module will be denoted $\mathcal{M}^f$. Its states are tuples $\langle u_1, \ldots, u_h \rangle$, $h \geq 1$, whose last component u_h is a node, while all the other are boxes. Moreover, each u_j belongs to the $\mathcal{M}_i$ which the box u_{j-1} refers to. A state in the flat module is either a system or environment state depending on whether u_h is a system or an environment node, and also the propositional labeling of the state is determined by the labeling of u_h.

Now we proceed to a formal definition of expansion of a hierarchical module $\mathcal{M} = \langle \mathcal{M}_1, \ldots, \mathcal{M}_n \rangle$. For each component $\mathcal{M}_i$, we define the module $\mathcal{M}_i^f = (AP, S_i^f, E_i^f, R_i^f, in_i^f, L_i^f)$ as the expanded structure of $\mathcal{M}_i$ obtained as follows:

– $in_i^f = \langle in_i \rangle$;
– The set S_i^f (resp., E_i^f) of system (resp., environment) nodes of $\mathcal{M}_i^f$ is defined inductively:
  • if u is a system (resp., environment) node of $\mathcal{M}_i$ then $\langle u \rangle$ belongs to S_i^f (resp., E_i^f);
  • if u is a box of $\mathcal{M}_i$ with $Y_i(u) = j$, and $\langle u_1, \ldots, u_h \rangle$ is a system (resp., environment) state of $\mathcal{M}_j^f$, where $h \geq 1$, then $\langle u, u_1, \ldots, u_h \rangle$ belongs to S_i^f (resp., E_i^f).
– The transition relation R_i^f of $\mathcal{M}_i^f$ is defined inductively as follows:
  • for $(u, v) \in R_i$, if the sink v is a node then $(\langle u \rangle, \langle v \rangle) \in R_i^f$, and if v is a box with $Y_i(v) = j$ then $(\langle u \rangle, \langle v, in_j \rangle) \in R_i^f$;
  • if w is a box of $\mathcal{M}_i$ with $Y_i(w) = j$, and $(\langle u_1, \ldots, u_h \rangle, \langle v_1, \ldots, v_{h'} \rangle)$ is a transition of $\mathcal{M}_j^f$, for $h, h' \geq 1$, then $(\langle w, u_1, \ldots, u_h \rangle, \langle w, v_1, \ldots, v_{h'} \rangle)$ belongs to R_i^f.
– The labeling function $L_i^f : W_i^f \to 2^{AP}$ of $\mathcal{M}_i^f$ (where $W_i^f = S_i^f \cup E_i^f$) is defined inductively as follows:
  • if w is a node of $\mathcal{M}_i$, then $L_i^f(\langle w \rangle) = L_i(w)$;
  • if $w = \langle u, u_1, \ldots, u_h \rangle$, where $h \geq 1$, and u is a box of $\mathcal{M}_i$ with $Y_i(u) = j$, then $L_i^f(w) = L_j^f(\langle u_1, \ldots, u_h \rangle)$.

The module $\mathcal{M}_1^f$ is the expanded structure of $\mathcal{M}$ and therefore we just indicate it as $\mathcal{M}^f$ in the following.

The size $|\mathcal{M}_i|$ of $\mathcal{M}_i$ is the sum of $|W_i|$, $|Box_i|$, and $|R_i|$. The size of the hierarchical module $\mathcal{M}$ is the sum of the sizes of all $\mathcal{M}_i$. The nesting depth of $\mathcal{M}$, denoted $nd(\mathcal{M})$, is the length of the longest chain $i_1, i_2, \ldots, i_j$ of indices such that a box of $\mathcal{M}_{i_l}$ is mapped to i_{l+1}. Observe that each state of the expanded structure is a vector of length at most the nesting depth, and the size of the expanded module $\mathcal{M}^f$ can be exponential in the nesting depth, and is $O(|\mathcal{M}|^{nd(\mathcal{M})})$.

The *hierarchical module checking problem* for *CTL* is to decide for a given hierarchical module $\mathcal{M}$ and a *CTL* formula φ, whether $\mathcal{M}^f \models_r \varphi$.

As noted above, the last component of every state is a node (all the others being boxes), and the system or environment nature of the last component as well as its propositional labeling determines the nature and the propositional labeling of the entire state, respectively.

In the following sections we will consider the cases of hierarchical module *single-exit* (all the O_i contain just element), or *multiple-exit* and the special case of hierarchical module with a constant number of exit-nodes (*constant-exit*).

3 Tree Automata

In order to solve the program complexity of the hierarchical module checking problem for *CTL*, we use an automata theoretic approach; in particular, we exploit the formalisms of *Nondeterministic Büchi Tree Automata* (*NBT*) [Rab70, VW86] and introduce *Hierarchical Nondeterministic Büchi Tree Automata* (*HNBT*), that is *NBT* where states can be either ordinary node states or box states, which are tree automata themselves. Analogously to hierarchical modules, we consider both the cases single- or multiple-exit *HNBT*. *HNBT* extend to infinite trees the notion of hierarchical automata introduced in [ABE+05] on infinite words.

Nondeterministic Büchi Tree Automata (*NBT*). Here, we briefly recall the definition of *NBT* over complete k-ary trees, for a given $k \geq 1$. Formally, a *NBT* is a tuple $\mathcal{A} = (\Sigma, Q, in, \delta, \mathcal{F})$, where Σ is a finite input alphabet, Q and in are as in modules and they represent a finite set of states, and an initial state, respectively; $\delta : Q \times \Sigma \to 2^{Q^k}$ is a transition function, and $\mathcal{F} \subseteq Q$ is a Büchi acceptance condition.

Intuitively, when the automaton is in state q, reading an input node x labeled by $\sigma \in \Sigma$, then the automaton chooses a tuple $(q_1, \ldots, q_k) \in \delta(q, \sigma)$ and splits in k copies such that for each $1 \leq i \leq k$, a copy in state q_i is sent to the node $x \cdot i$ in the input tree.

A run of $\mathcal{A}$ on a Σ-labeled k-ary tree (T, V) (where $T = \{1, \ldots, k\}^*$) is a Q-labeled tree (T, r) such that $r(\varepsilon) = in$ and for each $x \in T$, we have that $(r(x \cdot 1), \ldots, r(x \cdot k)) \in \delta(r(x), V(x))$. For a path $\pi \subseteq T$, let $inf_r(\pi) \subseteq Q$ be the set of states that appear as the labels of infinitely many nodes in π. For a Büchi condition $\mathcal{F} \subseteq Q$, π is *accepting* if $inf_r(\pi) \cap \mathcal{F} \neq \emptyset$. A run (T, r) is *accepting* if all its paths are accepting. The automaton $\mathcal{A}$ accepts an input tree (T, V) iff there

is an accepting run of $\mathcal{A}$ over (T, V). The language of $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$, is the set of Σ-labeled (complete) k-ary trees accepted by $\mathcal{A}$. The emptiness problem for $\mathcal{A}$ is to check whether $\mathcal{L}(\mathcal{A}) = \emptyset$. The *size* $|\mathcal{A}|$ of an *NBT* $\mathcal{A}$ is $|Q| + |\delta|$, note that $|\delta|$ is at most $|\Sigma| \cdot |Q|^{k+1}$.

It is well-known that formulas of *CTL* can be translated into equivalent tree automata (accepting the models of the given formula). In particular, given a *CTL* formula φ one can construct an *NBT* over k-ary trees, for some degree $k \geq 1$, having as number of states (independent from k) $2^{O(|\varphi| \log(|\varphi|))}$, as stated in the following lemma.

Lemma 1 ([KVW00, Var98])
Given a CTL *formula* φ *over* AP *and* $k \geq 1$*, one can construct a* NBT $\mathcal{A}_\varphi$ *with number of states* $2^{O(|\varphi| \log |\varphi|)}$ *(independent from* k*) that accepts exactly the set of* 2^{AP}*-labeled complete* k*-ary trees that satisfy* φ*.*

Hierarchical Nondeterministic Büchi Tree Automata(*HNBT*). We now introduce *HNBT* over complete k-ary trees (for a given $k \geq 1$), as a hierarchical extension of *NBT*. An *HNBT* $\mathcal{A}$ over an alphabet Σ is a tuple $\langle \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$, with each $\mathcal{A}_i$ such that $\mathcal{A}_i = (\Sigma, W_i, \delta_i, Box_i, in_i, O_i, Y_i, \mathcal{F}_i)$ where W_i, Box_i, in_i, O_i, and Y_i are as in the components of hierarchical modules, $\mathcal{F}_i \subseteq W_i$ is a set of accepting states, and $\delta_i : (W_i \cup (Box_i \times \bigcup_{j>i} O_j)) \times \Sigma \to 2^{(W_i \cup Box_i)^k}$. A tuple $(q_1, \ldots, q_k)$ is in $\delta_i(q, \sigma)$ only if either $q \in W_i$ or q is a pair (b, s), where $b \in Box_i$ with $Y_i(b) = j$ and s is an exit-node of $\mathcal{A}_j$. Moreover, each q_i can be either a node state or a box state of $\mathcal{A}_i$.

The *size* $|\mathcal{A}_i|$ is $|W_i| + |Box_i| + |\delta_i|$, note that $|\delta_i|$ is at most $|\Sigma| \cdot (|W_i| + |Box_i|)^{k+2}$. The size of $\mathcal{A}$ is the sum of the sizes of all $\mathcal{A}_i$. Similarly to hierarchical modules, we can flat a *HNBT* $\mathcal{A}$ into an *NBT* $\mathcal{A}^f$ by defining the NBT's $\mathcal{A}_i^f$ similarly to what has been done for $\mathcal{M}_i^f$. Thus a state of $\mathcal{A}_i^f$ is a tuple consisting of all box states and having necessarily as last component a node state. A state of $\mathcal{A}_i^f$ is final if its last component is in $\mathcal{F}_j$, with $j \geq i$. A tree (T, V) is accepted by a *HNBT* $\mathcal{A}$ if there is an accepting run of $\mathcal{A}^f$ on (T, V). The language $\mathcal{L}(\mathcal{A})$ accepted by $\mathcal{A}$ is the set of the accepted trees.

To exploit the automata theoretic approach for the module checking problem for hierarchical module, we solve the emptiness problem for *HNBT*.

Lemma 2. *The emptiness problem for a single-exit* HNBT *is in* PTIME*. The emptiness problem for a multiple-exit* HNBT *is in* PSPACE*.*

Proof (sketch). For an *NBT* $\mathcal{A}$, one can check in polynomial time its emptiness by simply checking whether there exists in $\mathcal{A}$ a set G of "good" final states which is reachable by itself (i.e., for each state w of G there is a run that contains a subtree starting from w and whose frontier is contained in G) and that from the initial state of $\mathcal{A}$ it is possible to reach G [Rab70, VW86].

We now prove that also in the case of single-exit *HNBT* we can check emptiness in polynomial time, by opportunely embedding a component-wise exploration of the hierarchical automaton into the above *NBT*'s emptiness algorithm.

In fact, for a single-exit *HNBT* $\mathcal{A} = \langle \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$ instead of checking emptiness for the flat $\mathcal{A}_1^f$ we will use a simple property (checkable in polynomial

time) on suitable NBTs $\widehat{\mathcal{A}}_i$ and sets $\widehat{Box}_i$, constructed from $\mathcal{A}_i$. Let us for the moment give a non-constructive definition of these latter sets: for each i, let $\widehat{Box}_i = \{b \in Box_i \mid Y_i(b) = j,\ \mathcal{L}(\mathcal{A}_j^f) \neq \emptyset\}$. Moreover, let each $\mathcal{A}_i = (\Sigma, W_i, \delta_i, Box_i, ini_i, O_i, Y_i, \mathcal{F}_i)$, the *NBT* $\widehat{\mathcal{A}}_i$ is a tuple $(\Sigma, \widehat{Q}_i, \widehat{in}_i, \widehat{\delta}_i, \widehat{\mathcal{F}}_i)$, where

- $\widehat{Q}_i \subseteq W_i \cup Box_i$ and $W_i \subseteq \widehat{Q}_i$;
- $\widehat{\mathcal{F}}_i \subseteq \mathcal{F}_i \cup Box_i$ and $\mathcal{F}_i \subseteq \widehat{\mathcal{F}}_i$;

To each $\widehat{\mathcal{A}}_i$ can be associated a set of non complete k-ary trees, possibly having some finite paths, which can be seen, roughly speaking, as accepted by a Tree Automaton (not a NBT) with the following acceptance conditions: on the finite paths the acceptance is obtained considering the states of $\widehat{Box}_i$ as final states, while the infinite paths are accepted with the usual Büchi condition $\widehat{\mathcal{F}}_i$. To check the emptiness for the given *HNBT*, we will check whether such set of trees for $\widehat{\mathcal{A}}_1$ is empty. For each $\widehat{\mathcal{A}}_i$ we distinguish three different kinds of paths π of its runs, all starting from $\widehat{in}_i$:

A. π either is infinite and goes through a state of $\widehat{\mathcal{F}}_i$ infinitely often or is finite and its last state belongs to $\widehat{Box}_i$.
B. π is a finite path whose last state is in O_i (recall that $O_i \subseteq \widehat{Q}_i$, since $W_i \subseteq \widehat{Q}_i$) and π does not contain any state of $\widehat{\mathcal{F}}_i$.
C. π is a finite path whose last state is in O_i and it does contain at least one state of $\widehat{\mathcal{F}}_i$.

Let us now define inductively and bottom-up all $\widehat{\mathcal{A}}_i$ and the $\widehat{Box}_i$. For the base, let $\widehat{\mathcal{A}}_n = \mathcal{A}_n$ and $\widehat{Box}_n = \emptyset$. Suppose now that for $1 \leq i \leq n$ we have already defined all $\widehat{\mathcal{A}}_j$, for $j > i$. The sets $\widehat{Q}_i$ and $\widehat{\mathcal{F}}_i$ contain W_i and $\mathcal{F}_i$, respectively and, given a box $b \in Box_i$ such that $Y_i(b) = j$, we have that:

- if there exists a run of $\widehat{\mathcal{A}}_j$ containing at least a type-**B** path and all the others are either type-**A** paths or type-**C** paths, then $b \in \widehat{Q}_i$.
- if there exists a run of $\widehat{\mathcal{A}}_j$ containing at least a type-**C** path and all the remaining are of type-**A** paths, then $b \in \widehat{Q}_i$ and $b \in \widehat{\mathcal{F}}_i$.
- if there exists a run of $\widehat{\mathcal{A}}_j$ whose all paths are of type-**A** paths, then $b \in \widehat{Box}_i$ and $b \in \widehat{Q}_i$.

Moreover $\widehat{in}_i = in_i$ and

- for $q \in W_i$, $\widehat{\delta}_i(q, \sigma) = \delta_i(q, \sigma)$ and
- for $q \in Box_i \cap \widehat{Q}_i$, $\widehat{\delta}_i(q, \sigma) = \delta_i((q, s), \sigma)$ with $s \in O_i$.

Observe that the set $\widehat{Box}_i$ contains a box $b \in Box_i$ if it appears in an accepting run of $\mathcal{A}_i^f$. Moreover if there exists a run of $\widehat{\mathcal{A}}_j$ containing a type-**C** path π, then this run can be taken infinitely often in an accepting run (T, r) of $\mathcal{A}_i^f$: in this way, in fact, the final state of $\widehat{\mathcal{A}}_j$ occurring in π, appears infinitely often in some paths of (T, r). Thus we consider b as a final state of $\widehat{\mathcal{A}}_i$.

Now it is easy to see that a tree (T, V) is accepted by $\mathcal{A}_i^f$ if and only if there exists a run of $\widehat{\mathcal{A}}_i$ whose paths are all of type-**A** . Observe that the set $\widehat{Box}_i$ is now defined constructively and is consistent with the previous definition.

Now since $\mathcal{L}(\mathcal{A}) \neq \emptyset$ if and only if there exists a tree (T, V) accepted by $\mathcal{A}_1^f$, we have that an algorithm to solve the emptiness problem can be easily given, based on the construction of $\widehat{\mathcal{A}}_i$'s. Actually it remains to be convinced that the existence of the runs required in those constructions can be done in polynomial time. This can be accomplished by using a fixed-point algorithm which resembles the one given by Rabin in [Rab70] for *NBT*. In an extended version of the paper we will give the full details of such algorithm.

Consider now a multiple-exit *HNBT* A. We now sketch a nondeterministic algorithm running in polynomial space, and from Savitch's theorem, we get our result. As in the single-exit case, we will construct some NBT $\widehat{\mathcal{A}}_i$'s, but now the construction is accomplished nondeterministically and in a top-down way, starting from $i = 1$. To obtain each $\widehat{\mathcal{A}}_i$, the algorithm, for each box $b \in Box_i$ with $Y_i(b) = j$, either guesses that $\mathcal{L}(\mathcal{A}_j^f) \neq \emptyset$ or chooses two (possibly empty) sets $X \subseteq O_j$ and $Y \subseteq O_j$ and guesses, for some $s \in O_j$, that there exists a run in $\widehat{\mathcal{A}}_j$ having type-**B** paths from $\widehat{in}_j$ to $s \in X$ and type-**C** paths from $\widehat{in}_j$ to $s \in Y$. According to these choices, the NBT $\widehat{\mathcal{A}}_i$ is constructed, similarly as in the case of single-exit. Then the algorithm proceeds by checking the guessed property of $\widehat{\mathcal{A}}_{Y_i(b)}$, for each $b \in Box_i$. In this step $\widehat{\mathcal{A}}_{Y_i(b)}$ is constructed, and this obviously, implies other guesses for the boxes belonging to $\mathcal{A}_{Y_i(b)}$. This chain of guesses naturally ends when $\widehat{\mathcal{A}}_n$ has to be constructed, since Box_n is empty. Observe that the overall space necessary during the execution of the algorithm does not exceed the size of $\mathcal{A}$ and this concludes the proof. □

We now conclude this section by showing that the above results are also tight. For the single-exit case, notice that *NBT* are a special case of *HNBT* and for *NBT* the emptiness problem is already known to be PTIME-hard.

For the multiple-exit case, we use a polynomial reduction from the model-checking problem for multiple-exit hierarchical state machines w.r.t constant *CTL* formulas, which is known to be PSPACE-hard [AY01]. In order to apply this reduction, we have first to define a *cross product* between an *HNBT* $\mathcal{A}'$ and an *NBT* $\mathcal{A}''$, say it $\mathcal{A}' \otimes \mathcal{A}''$, that allows to construct in polynomial time an *HNBT* whose flattening is equivalent to the Cartesian product of the *NBT*'s $\mathcal{A}'^f$ and $\mathcal{A}''$. We now formally show how to construct $\mathcal{A}' \otimes \mathcal{A}''$.

Let $\mathcal{A}' = \langle \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$ be an *HNBT*, with $\mathcal{A}_i = (\Sigma, W_i, \delta_i, Box_i, in_i, O_i, Y_i, \mathcal{F}_i)$, and $\mathcal{A}'' = (\Sigma, Q, in, \delta, \mathcal{F}'')$ with $Q = \{q_1, \ldots, q_m\}$ and $in = q_1$. The product $\mathcal{A}' \otimes \mathcal{A}''$ is the *HNBT* $\mathcal{A} = \langle \mathcal{A}_{11}, \ldots, \mathcal{A}_{1m}, \ldots, \mathcal{A}_{n1}, \ldots, \mathcal{A}_{nm} \rangle$, where each component $\mathcal{A}_{ij} = (\Sigma, W_i \times Q, \delta_{ij}, Box_i \times Q, (in_i, q_j), O_i \times Q, Y_{ij}, \mathcal{F}_{ij})$, $1 \leq i \leq m$ and $1 \leq j \leq n$, is such that

- $\mathcal{F}_{ij} = \mathcal{F}_i \times \mathcal{F}''$
- $Y_{ij}(b, q) = m(i' - 1) + j'$ if $Y_i(b) = i'$ and $q = q_{j'}$,
- if $(q_1'', \ldots, q_k'') \in \delta(q'', \sigma)$ and $(q_1', \ldots, q_k') \in \delta_i(q', \sigma)$ then
  - if $q' \in W_i$ then $((q_1', q_1''), \ldots, (q_k', q_k'')) \in \delta_{ij}((q', q''), \sigma)$
  - if $q' = (b, s)$, with $b \in Box_i$ and $s \in O_{Y_i(b)}$, then $((q_1', q_1''), \ldots, (q_k', q_k'')) \in \delta_{ij}((b', s'), \sigma)$, where $b' = (b, q)$, for some $q \in Q$, and $s' = (s, q'')$)

Lemma 3. *Given an* HNBT $\mathcal{A}'$ *and an* NBT $\mathcal{A}''$*, the* HNBT $\mathcal{A} = \mathcal{A}' \otimes \mathcal{A}''$ *accepts the language* $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}') \cap \mathcal{L}(\mathcal{A}'')$ *and has size* $O(|Q|^2 \cdot |\mathcal{A}'| \cdot |\mathcal{A}''|)$.

Proof. Let $\mathcal{A}'$ be an *HNBT*, $\mathcal{A}''$ be an *NBT* and $\mathcal{A} = \mathcal{A}' \otimes \mathcal{A}''$, as described above, with components $\mathcal{A}_{ij}$. Since $\mathcal{A}^f$ is not isomorphic to $\mathcal{A}'^f \times \mathcal{A}''$, we prove the lemma by showing an isomorphism among the run of $\mathcal{A}^f$ and those of $\mathcal{A}'^f \times \mathcal{A}''$.

Given a run (T, r) of $\mathcal{A}^f$, we can define a run (T, r') of $\mathcal{A}'^f$ and a run (T, r'') of $\mathcal{A}''$ as follows. The run (T, r') is obtained by projecting for each state $\langle (q'_1, q''_1), \ldots, (q'_h, q''_h) \rangle$ in (T, r) the first components, thus getting the state $\langle q'_1, \ldots, q'_h \rangle$ of $\mathcal{A}'^f$. The run (T, r'') of $\mathcal{A}''$ is obtained by projecting the second component of the node in each state (recall that only (q'_h, q''_h) is a node, while all the other are boxes). Symmetrically, given the two runs, we can define (T, r) of $\mathcal{A}^f$. Now it immediately follows that (T, r) is accepting if and only if (T, r') and (T, r'') are both accepting runs. Thus the accepted languages is $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}') \cap \mathcal{L}(\mathcal{A}'')$.

Consider now the size of each component $\mathcal{A}_{ij}$: from the definition of the cross product $\otimes$, each component $\mathcal{A}_{ij}$ is obtained by pairing the initial node in_i of $\mathcal{A}_i$ with q_j. The number of the states is $|W_i| \cdot |Q|$ and the number of superstates is $|Box_i| \cdot |Q|$. The size $|\delta_{ij}|$ of the transition function is bounded by $|\delta_i| \cdot |\delta| \cdot |Q|$, since when the transition is defined on a pair (b, s) then b can be paired with any state $q \in Q$. Thus the overall size of $\mathcal{A} = \mathcal{A}' \otimes \mathcal{A}''$ is $O(|Q|^2 \cdot |\mathcal{A}'| \cdot |\mathcal{A}''|)$. □

Let us now turn back to the desired reduction. Let φ be a fixed *CTL* formula and $\mathcal{M}$ be a hierarchical closed state machine with multiple exits. Let $\mathcal{A}_\mathcal{M}$ be an *HNBT* obtained from $\mathcal{M}$ by considering all its states as accepting and collecting all its relations in "tree-like" transitions (observe that $\mathcal{A}_\mathcal{M}$ suffices to be deterministic). More formally, for each $\mathcal{M}_i = (AP, S_i, E_i, R_i, Box_i, O_i, in_i, L_i, Y_i)$, in $\mathcal{M}$ we add to $\mathcal{A}_\mathcal{M}$ the component $\mathcal{A}_i = (2^{AP}, W_i, \delta_i, Box_i, in_i, O_i, Y_i, \mathcal{F}_i)$, where $W_i = S_i \cup E_i$, $\mathcal{F}_i = W_i$, and $\delta_i(w, L_i(w)) = \{(w_i, \ldots, w_d)\}$ iff, for each $1 \leq j \leq d$, $(w, w_d) \in R_i$. By Lemma 1, we can construct an *NBT* $\mathcal{A}_\varphi$ accepting all models of φ. Note that $\mathcal{A}_\varphi$ has a fixed size, since φ is also fixed. By Lemma 3, we can construct in polynomial time an *HNBT* $\mathcal{A}_{\mathcal{M} \otimes \varphi}$ accepting the intersection[1] of $\mathcal{A}_K$ and $\mathcal{A}_\varphi$. Clearly, K satisfies φ iff $\mathcal{L}(\mathcal{A}_{K \otimes \varphi}) \neq \emptyset$. This, together with Lemma 2 leads to the desired result.

Theorem 1. *(*i*) The emptiness problem for a single-exit* HNBT *is* Ptime*-complete. (*ii*) The emptiness problem for a multiple-exit* HNBT *is* Pspace*-complete.*

4 Deciding Hierarchical Module Checking

In this section, we solve the program complexity of the *CTL* hierarchical module checking problem. In particular, we show that this problem is in Ptime for

[1] One can observe that $\mathcal{A}_\mathcal{M}$ may not be complete and that $\mathcal{A}_\mathcal{M}$ and $\mathcal{A}_\varphi$ may disagree on the number of node successors. It is not hard to see that, by duplicating successor states, we can adapt the previous constructions in order to obtain $\mathcal{A}_\mathcal{M}$ and $\mathcal{A}_\varphi$ as k-ary complete automata.

single-exit modules and in PSPACE in the multiple-exit case. By recalling that the program complexity for the classical *CTL* module checking is PTIME-hard and the program complexity for the *CTL* hierarchical model checking is PSPACE-hard, we get that our results are also tight.

Our solution to both problems is based on an automata–theoretic approach, by extending an idea of [KVW01]. In practice, we take into account that the input module is hierarchical. Therefore, we extend [KVW01]'s idea to each component of the module and use the automata product $\otimes$ introduced in the previous section, instead of a classical Cartesian product. In more details, let $\mathcal{M}$ be a hierarchial module and φ a fixed *CTL* formula. We decide the module checking problem for $\mathcal{M}$ against φ by building an *HNBT* $\mathcal{A}_{\mathcal{M}\otimes\neg\varphi}$ as $\mathcal{A}_{\mathcal{M}} \otimes \mathcal{A}_{\neg\varphi}$. Essentially, the first automaton, $\mathcal{A}_{\mathcal{M}}$, is an *HNBT* that accepts trees of $exec(\mathcal{M}^f)$, and the second automaton is an *NBT* $\mathcal{A}_{\neg\varphi}$ that accepts all trees that do not satisfy φ. Thus, $\mathcal{M} \models_r \varphi$ iff $\mathcal{L}(\mathcal{A}_{\mathcal{M}\otimes\neg\varphi})$ is empty. Now, recall from Lemma 3 that the cross product between $\mathcal{A}_{\mathcal{M}}$ and $\mathcal{A}_{\neg\varphi}$ corresponds to an *HNBT* (which can be constructed in polynomial time) whose flattening is equivalent to the Cartesian product of the *NBT*'s $\mathcal{A}^f_{\mathcal{M}}$ and $\mathcal{A}_{\neg\varphi}$. The component automata of the obtained *HNBT* will have a number of exit nodes that depends on the number of states of $\mathcal{A}_{\neg\varphi}$, which in turn depends, by Lemma 1, on the size of the formula φ. Since here we are interested on the program complexity of the hierarchical module checking problem, we assume the formula to be fixed. Therefore, $\mathcal{A}_{\mathcal{M}\otimes\neg\varphi}$ will have multiple exits iff $\mathcal{A}_{\mathcal{M}}$ does, and a constant number of exit nodes, otherwise.

Let us now discuss about the emptiness problem for *HNBT*'s with a constant number of exits. That is, we are interested in determining the complexity of the emptiness problem for the set $\{\mathcal{A} \mid \mathcal{A}$ is an *HNBT* with at most d-exit nodes$\}$, for a fixed natural number d. First observe that in the algorithm we have proposed in Lemma 2 for checking the emptiness of *HNBT*'s with single exits, each box either contributes to check the existence of an accepting run or not at all. On the opposite, in the multiple-exit case, we have to remember for each box which exit node ensures acceptance and which does not. Therefore, for each box some splitting may be required. For instance consider a component $\mathcal{A}_i$ with two exit nodes w_1 and w_2. It may be that a run exits in w_1 trough paths all visiting at least a final state and in w_2 trough a path that does not. Thus, we need to split $\mathcal{A}_i$ into four copies, depending whether both, only w_1, only w_2, or none can be considered in the extended set of final states. In general, if we start in Lemma 2 with an *HNBT* having at most d exit nodes, we need to generate 2^d copies of each component automaton, in the worst case. Since d is a fixed parameter, it turns out that the emptiness problem also for this automata remains in PTIME as reported in the following proposition.

Proposition 1. *The emptiness problem for a constant-exit* HNBT *is in* PTIME.

To conclude with our idea of solving the program complexity for *CTL* hierarchical module checking let us give some details on how to construct $\mathcal{A}_{\mathcal{M}}$ for a hierarchical module $\mathcal{M} = \langle \mathcal{M}_1, \ldots, \mathcal{M}_n \rangle$, with each $\mathcal{M}_i = (AP, S_i, E_i, R_i, Box_i, O_i, in_i, L_i, Y_i)$. First, we recall that each component automaton of $\mathcal{A}_{\mathcal{M}}$ can only work

on complete k-ary trees, while trees in $exec(M^f)$ may be not. To overcome this problem, we consider an equivalent representation of $exec(M^f)$ in which all nodes have degree $k = max\{bd(w) \mid w \in \bigcup_i S_i \cup E_i\}$, where $bd(w)$ denotes the branching degree of w (i.e. the number of its successors). Now, recall that each tree in $exec(M^f)$ is a 2^{AP}-labeled tree that is obtained from (T_{M^f}, V_{M^f}) by suitably pruning some of its subtrees. We can encode the tree (T_{M^f}, V_{M^f}) as a $2^{AP} \cup \{\bot\}$-labeled complete k-ary tree (where $\bot$ is a fresh atomic proposition not belonging to AP) in the following way: for each node $x \in T_M$ with d children $(x \cdot 1, \ldots, x \cdot d)$ (note that $1 \leq d \leq k$ as R_i is total), we add the children $(x \cdot (d+1), \ldots, x \cdot k)$ and label these new nodes with $\bot$; finally, for each node x labeled by $\bot$ we add recursively k-children labeled by $\bot$. Let $(\{1, \ldots, k\}^*, V')$ be the tree thus obtained. Then, we can encode a tree $(T, V) \in exec(M^f)$ as the $2^{AP} \cup \{\bot\}$-labeled complete k-ary tree obtained from $(\{1, \ldots, k\}^*, V')$ preserving all the labels of nodes of $(\{1, \ldots, k\}^*, V')$ that either are labeled by $\bot$ or belong to T, and replacing all the labels of nodes (together with the labels of the corresponding subtrees) pruned in (T, V) with the label $\bot$. In this way, all the trees in $exec(M^f)$ have the same structure (they all coincide with $\{1, \ldots, k\}^*$), and they differ only in their labeling. Thus, the proposition $\bot$ is used to denote both "disabled" nodes and "completion" nodes[2]. Moreover, since we consider environments that do not block the system, for each node associated with an enabled environment node, at least one successor is not labeled by $\bot$. Let us denote by $\widehat{exec}(\mathcal{M})$ the set of all $2^{AP} \cup \{\bot\}$-labeled k-ary trees obtained from $(\{1, \ldots, k\}^*, V')$ in the above described manner. We now show an *HNBT* $\mathcal{A}_{\mathcal{M}}$ accepting $\widehat{exec}(\mathcal{M})$. $\mathcal{A}_{\mathcal{M}}$ is the tuple $\langle \mathcal{A}_{(1,\top)}, \mathcal{A}_{(2,\top)}, \mathcal{A}_{(2,\bot)}, \mathcal{A}_{(2,\vdash)}, \ldots, \mathcal{A}_{(n,\top)}, \mathcal{A}_{(n,\bot)}, \mathcal{A}_{(n,\vdash)} \rangle$, where for $1 \leq i \leq n$ and $x \in \{\bot, \top, \vdash\}$, each $\mathcal{A}_{(i,x)} = \langle \Sigma, W_i', \delta_i, Box_i', (in_i, x), O_i', Y_i', W_i \rangle$ is defined as follows (recall $W_i = S_i \cup E_i$):

- $\Sigma = 2^{AP} \cup \{\bot\}$;
- $W_i' = W_i \times \{\bot, \top, \vdash\}$. The automaton $\mathcal{A}_{(i,x)}$ starts from (in_i, x). For example, the computation starts from $(in_i, \bot)$ whenever a box corresponding to $\mathcal{A}_i$ has been disabled. From states of the form $(w, \bot)$, $\mathcal{A}_{(i,x)}$ can read only the letter $\bot$, from states of the form $(w, \top)$, it can read only letters in 2^{AP}. Finally, when $\mathcal{A}_{(i,x)}$ is in state $(w, \vdash)$, then it can read both letters in 2^{AP} and the letter $\bot$. In this last case, it is left to the environment to decide whether the transition to a state of the form $(w, \vdash)$ is enabled. The three types of states are used to ensure that the environment enables all transitions from enabled system nodes, enables at least one transition from each enabled environment node, and disables transitions from disabled nodes.
- $O_i' = O_i \times \{\bot, \top, \vdash\}$. Clearly, we can have three types of exit nodes.
- $Box_i' = Box_i \times \{\bot, \top, \vdash\}$. As for states, we can have three types of boxes, which are used to ensure that, regarding the initial node w of the component automaton corresponding to a box, the environment can enable all transitions from w whenever w is an enabled system node, enable at least one

[2] As stated in [KVW01], the use of the atomic proposition $\bot$ must be taken into account while building $\mathcal{A}_{\neg\varphi}$. This can be easily handled by opportunely modifying the formula φ by exploiting an argument similar to that used in [KVW01].

transition from w whenever w is an enabled environment node, and disables transitions from w whenever w is a disabled node.

- Y_i' is such that $Y_i'(b,x) = Y_i(b), x$. That is, from a box (b,x), $\mathcal{A}_i$ calls the automaton $\mathcal{A}_{(Y_i(b),x)}$. Then the computation continues from $(in_{Y_i(b)}, x)$ as described above.
- The transition function $\delta_i : (W_i' \cup (Box_i' \times \bigcup_{j>i} O_j')) \times \Sigma \to 2^{(W_i' \cup Box_i')^k}$ is defined as follows. Let $z = w \in W_i$ or $w \in O_j$ such that $z = (b,w) \in (Box_i \times \bigcup_{j>i} O_j)$ and $Y_i(b) = j$. Let $succ(z) = (z_1, \ldots, z_d)$, with $1 \leq d \leq k$, and for $m, m' \in \{\top, \vdash, \bot\}$, let z' be either (w,m), if $z = w$, or $z' = ((b,m'),(w,m))$, if $z = (b,w)$. Then, δ_i is as follows:
  - For $w \in W_i \cup O_j$ and $m \in \{\vdash, \bot\}$, we have

$$\delta_i(z', \bot) = \{(\underbrace{(z_1, \bot), \ldots, (z_d, \bot), (z, \bot), \ldots, (z, \bot)}_{k \text{ pairs}})\}$$

    That is, $\delta_i(z', \bot)$ contains exactly one k-tuple. In this case all the successors of the current node are disabled.
  - For $w \in S_i \cup (S_j \cap O_j)$ and $m \in \{\top, \vdash\}$ we have

$$\delta_i(z', L_i(w)) = \{(\underbrace{(z_1, \top), \ldots, (z_d, \top), (z, \bot), \ldots, (z, \bot)}_{k \text{ pairs}})\}$$

  - For $m \in \{\top, \vdash\}$ and either $w \in E_i$ and $g = w$ or $w \in (E_j \cap O_j)$ and $g = b$ we have $\delta_i(z', L_i(w)) =$

$$\begin{array}{c} \{ ((z_1, \top), (z_2, \vdash), \ldots, (z_d, \vdash), \ (g, \bot), \ldots, (g, \bot)), \\ ((z_1, \vdash), (z_2, \top), \ldots, (z_d, \vdash), \ (g, \bot), \ldots, (g, \bot)), \\ \vdots \qquad\qquad\qquad\qquad \vdots \\ ((z_1, \vdash), (z_2, \vdash), \ldots, (z_d, \top), \ (g, \bot), \ldots, (g, \bot))\}. \end{array}$$

    That is, $\delta_i(z', L_i(w))$ contains d k-tuples. When the automaton proceeds according to the i-th tuple, the environment can disable the transitions to all successors of the current state, except the transition associated with z_i, which must be enabled.

One can be convinced on the fact that $\mathcal{A}$ is polynomial in the size of $\mathcal{M}$, by noting that each $\mathcal{A}_{(i,x)}$ has $3 \cdot |\mathcal{M}_i|$ states, and $|\delta_i|$ is bounded by $O(k \cdot |R_i|)$. Thus, by summing up our idea trough the above polynomial construction of $\mathcal{A}_{\mathcal{M}}$, the exponential-constant time construction given by Lemma 1 for $\mathcal{A}_{\neg\varphi}$, Lemma 3, Proposition 1, and Lemma 2 the following result holds.

Theorem 2. *The program complexity for the* CTL *hierarchical module checking problem is Ptime-complete in the case of single-exit modules and Pspace-complete in in the case of multiple-exit modules.*

5 Conclusions

In this paper, we have introduced and solved the program complexity for the hierarchical module checking problem for *CTL*, both in the case of single-exit and multiple-exit modules. An immediate exponential solution can be obtained by flattening the hierarchical module and then apply the classical algorithm. By avoiding the flattening, we have shown algorithms having a better performance and, in particular, working not harder than those used in the closed hierarchical system case. As future directions, it would be worth to consider more involved scenarios both on the module checking side (e.g., pushdown module checking [BMP05], systems with incomplete information [AMV07]) and on the hierarchical side (e.g., recursive state machine [ABE+05], with both nodes and boxes labeled with atomic propositions [LNPP08]).

References

[ABE+05] Alur, R., Benedikt, M., Etessami, K., Godefroid, P., Reps, T.W., Yannakakis, M.: Analysis of recursive state machines. ACM Trans. Program. Lang. Syst. 27(4), 786–818 (2005)

[AMV07] Aminof, B., Murano, A., Vardi, M.Y.: Pushdown module checking with imperfect information. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 460–475. Springer, Heidelberg (2007)

[AY01] Alur, R., Yannakakis, M.: Model checking of hierarchical state machines. ACM Trans. Program. Lang. Syst. 23(3), 273–303 (2001)

[BMP05] Bozzelli, L., Murano, A., Peron, A.: Pushdown module checking. In: Sutcliffe, G., Voronkov, A. (eds.) LPAR 2005. LNCS, vol. 3835, pp. 504–518. Springer, Heidelberg (2005)

[CE81] Clarke, E.M., Emerson, E.A.: Design and verification of synchronization skeletons using branching time temporal logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1982)

[CGP99] Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)

[KVW00] Kupferman, O., Vardi, M.Y., Wolper, P.: An Automata-Theoretic Approach to Branching-Time Model Checking. J. of the ACM 47(2), 312–360 (2000)

[KVW01] Kupferman, O., Vardi, M.Y., Wolper, P.: Module Checking. Information and Computation 164(2), 322–344 (2001)

[LNPP08] LaTorre, S., Napoli, M., Parente, M., Parlato, G.: Verification of scope-dependent hierarchical state machines. Information and Computation 206(9,10), 1161–1177 (2008)

[QS81] Queille, J.P., Sifakis, J.: Specification and verification of concurrent programs in Cesar. In: Dezani-Ciancaglini, M., Montanari, U. (eds.) Programming 1982. LNCS, vol. 137, pp. 337–351. Springer, Heidelberg (1982)

[Rab70] Rabin, M.O.: Weakly definable relations and special automata. Mathematical Logic and Foundations of Set theory (1970)

[Var98] Vardi, M.Y.: Reasoning about the past with two-way automata. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443, pp. 628–641. Springer, Heidelberg (1998)

[VW86] Vardi, M.Y., Wolper, P.: Automata-theoretic techniques for modal logics of programs. J. of Computer and System Sciences 32(2), 182–221 (1986)

`Valigator`: A Verification Tool with Bound and Invariant Generation*

Thomas A. Henzinger, Thibaud Hottelier, and Laura Kovács

EPFL, Switzerland

Abstract. We describe `Valigator`, a software tool for imperative program verification that efficiently combines symbolic computation and automated reasoning in a uniform framework. The system offers support for automatically generating and proving verification conditions and, most importantly, for automatically inferring loop invariants and bound assertions by means of symbolic summation, Gröbner basis computation, and quantifier elimination. We present general principles of the implementation and illustrate them on examples.

1 Introduction

In [16], a framework for generating polynomial equations as loop invariants was presented for a rich class of so-called *P-solvable* loops with ignored loop conditions. Implemented in the software package `Aligator`, the approach was successfully tested on many examples. However, `Aligator` was not able to infer properties depending on the loop conditions.

In the current paper we address this problem and present `Valigator`, an automatic tool that extends and uses the functionalities of `Aligator`. More precisely, `Valigator` enables `Aligator` to infer stronger invariants involving polynomial equalities and inequalities by treating loop conditions. In addition, `Valigator` supports generating and proving verification conditions using the inferred loop invariants, and proving the partial correctness of programs annotated with pre- and postconditions. We consider programs containing loops with sequencing, nested conditionals, and assignments, and impose structural constraints on the type of assignments. We require that loop conditions are linear inequalities, variables from loop conditions are changed using affine mappings, and most importantly, branches in loops commute. By commuting we mean that variables in the loop condition are changed by the same affine mappings in all conditional branches.

The purpose of this paper is to discuss the underlying principles of `Valigator`, whose main features are as follows. (1) It contains a prototype verification condition generator based on the strongest-postcondition strategy [13]. (2) It integrates `Aligator` as its invariant inference engine and thus has access to a wealth of powerful algorithms from the computer algebra system `Mathematica` [24]. Moreover, we extended `Aligator` by implementing an approach for automatically inferring polynomial equalities and *inequalities* as invariants from the polynomial closed form of the loop by imposing *bound constraints* on the number of loop iterations. (3) Finally, `Valigator`

* This research was supported by the Swiss NSF.

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 333–342, 2008.

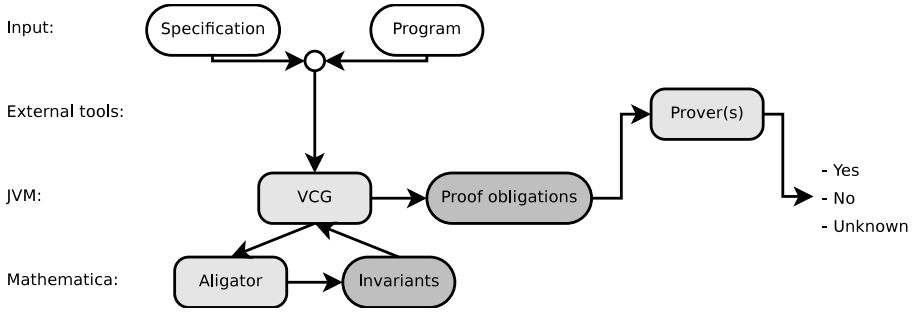


Fig. 1. The `Valigator` tool

uses the automated theorem proving tools `Z3`, `CVC3`, and `STP` [11, 5, 15] for proving the correctness of the verification conditions.

In the sequel, we will discuss in more detail the main ingredients of `Valigator`. We will present general principles of the implementation and illustrate them on examples. In order to improve readability, we present the input and output lines of `Valigator` commands in a simplified form.

Implementation and Installation. `Valigator` is implemented in the `Scala` programming language [20], that compiles to the `Java` bytecode. Beside exploiting the `Scala` capabilities, `Valigator` integrates `Scala` with the computer algebra system `Mathematica` in a transparent way by relying on the `JLink` toolkit [24]. Using and controlling the `Mathematica` kernel directly from a `Scala` program is thus supported in `Valigator`. The overall workflow of `Valigator` is illustrated in Figure 1.

`Valigator` is available at:

`http://mtc.epfl.ch/software-tools/Aligator/Valigator/`.

The current version of the source distribution is 0.1 and runs under most recent `Linux` versions.

Experiments. We have successfully tried our implementation on many examples; – see the mentioned `URL`. For each of the examples, the results were obtained in less than 5 seconds on a machine with a 2.0GHz CPU and 2GB of RAM. The most time-consuming part of `Valigator` lies in proving verification conditions, whereas the generation of verification conditions together with invariant and bound inference requires less than 2 seconds for all examples we tried.

Related Work. We only mention some of the numerous tools that are related to `Valigator`. Such systems include the static program verifiers `Esc/Java` [14], `LOOP` [23], and `JIVE` [18] for sequential `Java`. Inputs to these tools are `Java` source programs with user-supplied annotations expressed in `JML` [17]. Generating verification conditions, these systems are respectively connected with the theorem provers `Simplify` [12], `PVS` [21], and `Isabelle` [19] to verify proof obligations. It is well-known that producing first-order verification conditions requires loop invariants. These tools are thus powerful for programs whose invariants are specified by the user and, due to the nature of the employed provers, they support checking assertions over integers

only if they are linear. A similar verification environment has been developed in the KeY tool [1] for Java programs, and in Spark [2] for Ada code.

A closely related approach to Valigator is the Spec♯ [4] verifier for C♯ programs, which uses the Boogie [3] tool for verification. Boogie combines an invariant inference engine based on the abstract interpretation framework, a verification condition generator, and the theorem prover Z3 [11]. Inferred invariants over integers are again required to be linear, because Z3 handles only linear arithmetic.

The main difference between the mentioned systems and Valigator is that Valigator supports the *automatic generation of polynomial* invariants by means of symbolic computation and offers a richer choice of proof tools over integers for proving polynomial (and not just linear) verification conditions. However, many of the discussed tools handle data types such as arrays and pointers, which is not yet the case for Valigator.

Inferring polynomial invariants is also supported by the Polyinvar tool [22] by bounding a priori the degree of polynomials. Valigator imposes no such bounds, yet Polyinvar can handle more complex loops. We are also not aware of an integration of Polyinvar into a verification environment.

2 Valigator: System Description

Inputs to Valigator are programs formulated in a custom language similar to a subset of the programming language C, with sequencing, assignments, conditionals, and loops, involving addition and multiplication over integers, augmented with pre- and postconditions. Data types are not yet handled. The language uses assume and assert constructs for stating, respectively, pre- and postconditions. In the sequel we call an *annotated program* a program with a precondition and a postcondition.

We require that loops be P-solvable. Namely, (i) they contain only assignments to variables and nested conditionals; (ii) variables in assignments range over numeric types, such as integers; (iii) values of variables can be expressed as polynomials of the initial values of variables (those when the loop is entered), the loop counter, and some new variables, where there are algebraic dependencies among the new variables; (iv) assignments to variables X' that appear in the loop condition are affine mappings satisfying the matrix equation $X' = AX' + B$ with a square matrix A and column vector B of numeric constants, where the main diagonal of A contains only 1s; (v) conditional branches in loops commute, i.e variables X' from the loop conditions are changed in the same manner on each branch; (vi) and finally, the loop condition is expressed by linear inequalities over loop variables X'. Note that condition (iv) ensures the existence of polynomial closed forms of X' as univariate polynomials in the loop counter, whereas conditions (iv)-(vi) yield polynomial inequalities in the loop counter.

The syntax of considered P-solvable loops is thus as below.

$$\begin{aligned}&\texttt{while}\,(b_0)\,\{s_0;\ \texttt{if}(b_1)\,\texttt{then}\,\{s_1\}\\ &\qquad\texttt{else}\{\ldots\texttt{else}\{\texttt{if}(b_{k-1})\,\texttt{then}\,\{s_{k-1}\}\,\texttt{else}\,\{s_k\}\}\ldots\};s_{k+1}\}\end{aligned} \tag{1}$$

where $k \in \mathbb{N}$, $s_0, \ldots, s_{k+1}$ are sequences of assignments and $b_0, \ldots, b_{k-1}$ are boolean expressions.

Verification conditions are automatically derived using the strongest postcondition method and `Aligator`. Their validity is checked by state-of-the-art theorem provers.

Command 2.1 : `Valigator[C]`

Input: Annotated program C
Output: `Yes/No/Unknown`, where

- `Yes` means that all verification conditions were successfully proved, and hence the partial correctness of the input is established;
- `No` is returned when at least one verification condition was disproved, and hence either a bug in the program was found or the invariants generated by `Aligator` were not strong enough.
- `Unknown` is answered when at least one verification condition was disproved because, due to some unsafe arithmetic simplifications during invariant inference, the derived invariant is not actually a loop invariant (see Subsection 2.2).

In the last two cases, when the invariants generation fails, it is possible to manually give invariants to `Valigator` using annotations in the program source code.

Example 2.1. Consider the annotated program computing the sum of two integers a and b, such that b is non-negative.

```
Input: Valigator[assume (b >= 0);
                 res = a; cnt = b;
                 while(cnt > 0){cnt = cnt - 1; res = res + 1};
                 assert (res = a + b)]
Output: Yes
```

Verification of imperative programs in `Valigator` consists of invariant inference, generation and proving of verification conditions. In what follows, we discuss these steps in more detail, and summarize the main components of `Valigator` in Table 1.

Table 1. `Valigator` commands and their descriptions

`Valigator`: verification of programs with invariant and bound inference
Input: Annotated program Output: `Yes/No/Unknown`
`VCG`: verification condition generator
Input: Annotated program Output: List of verification conditions
`Aligator`: invariant generation for P-solvable loops with bound inference
Input: P-solvable loop and, optionally, initial values of loop variables Output: Loop invariant
`AnalyseBound`: bound analysis for P-solvable loops
Input: List of assignments over loop variables, loop condition and initial values of loop variables Output: Bound assertions over the values of loop variables

2.1 Generation of Verification Conditions

Generation of verification conditions is performed by the VCG command.

Command 2.2 : `VCG[C]`

Input: Annotated program C
Output: List of verification conditions (proof obligations)

VCG is a predicate transformer based on a list of inference rules. It treats the program structure recursively, statement-by-statement. Namely, VCG takes an annotated program, and repeatedly modifies the precondition such that at the end the program is "eliminated", and a logical formula is inferred as a collection of verification conditions. In fact, it computes the strongest postcondition in the abstract interpretation framework defined in [10].

For each assertion specified by `assert`, VCG generates a proof obligation. However, contrarily to other verification tools (see e.g. [2,4]), in Valigator we do not use `assert` for annotating loops with invariants. Instead, VCG invokes Aligator as its invariant inference engine whenever a loop is encountered. Invariant generation is thus part of VCG and the invariant inference takes advantage of the constant propagation performed by VCG. Using the invariant returned by Aligator, VCG generates two proof obligations corresponding to the initial and the inductiveness properties of the invariant.

Example 2.2. For Example 2.1, the verification conditions derived by VCG are below.

Input: $\mathsf{VCG}[C]$

Output:
$$b \geq 0 \ \wedge \ res = a \ \wedge \ cnt = b \ \Rightarrow \\ cnt + res = a + b \ \wedge \ (b > 0 \Rightarrow cnt \geq 0) \ \wedge \ (b \leq 0 \ \Rightarrow \ res = a \ \wedge \ cnt = b)$$

$$b \geq 0 \ \wedge \ cnt > 0 \ \wedge \ cnt + res = a + b \wedge \\ (b > 0 \Rightarrow cnt \geq 0) \ \wedge \ (b \leq 0 \ \Rightarrow \ res = a \ \wedge \ cnt = b) \ \Rightarrow \\ (cnt - 1) + (res + 1) = a + b \ \wedge \ (b > 0 \Rightarrow cnt - 1 \geq 0) \wedge \\ (b \leq 0 \ \Rightarrow \ res + 1 = a \ \wedge \ cnt - 1 = b)$$

$$b \geq 0 \ \wedge \ cnt \leq 0 \ \wedge \ cnt + res = a + b \wedge \\ (b > 0 \Rightarrow cnt \geq 0) \ \wedge \ (b \leq 0 \ \Rightarrow \ res = a \ \wedge \ cnt = b) \ \Rightarrow \\ res = a + b$$

where the generated loop invariant is $cnt + res = a + b \ \wedge \ (b > 0 \Rightarrow cnt \geq 0) \ \wedge \ (b \leq 0 \ \Rightarrow \ res = a \wedge cnt = b)$, as shown in Example 2.3, and C is the input of Valigator given in Example 2.1. The first two proof obligations ensure soundness of the invariant; the last one corresponds to the `assert` statement.

2.2 Invariant Inference

Let X denote the set of loop variables, X_0 the corresponding initial values (before entering the loop), and n the iteration counter of the loop.

Command 2.3 : `Aligator[PLoop, IniVal→list of assignments]`

Input: P-solvable loop `PLoop` as in (1) and, *optionally*, a list of assignments specifying the initial values X_0 of X
Output: Loop invariant $\bigwedge_i p_i(X) = 0 \wedge \varphi(X)$, where $\varphi(X)$ is a boolean combination of polynomial equalities $q_j(X) = 0$ and inequalities $r_s(X) \geq 0$, with $p_i, q_j, r_s \in \mathbb{K}[X]$, where $\mathbb{K}$ can be either $\mathbb{Z}$ or $\mathbb{Q}$

Example 2.3. The invariant returned by `Aligator` for Example 2.1 is given below.

$$\text{Input: } \texttt{Aligator}[\texttt{while}(cnt > 0)\{cnt = cnt - 1;\ res = res + 1\},\ \texttt{IniVal} \rightarrow \{res = a; cnt = b\}]$$

$$\text{Output: } cnt + res = a + b \wedge (b > 0 \Rightarrow cnt \geq 0) \wedge (b \leq 0 \Rightarrow res = a \wedge cnt = b)$$

Note that the initial values of cnt and res are extracted by `VCG` from Example 2.1 (i.e. from the loop precondition), and then fed into `Aligator`.

Example 2.4. To illustrate the power of invariant and bound inference, let us now consider a P-solvable loop with conditional branches as given below. Its invariant property returned by `Aligator` is as follows.

$$\text{Input: } \texttt{Aligator}[\texttt{while}(x + y < 10)\ \{\texttt{if}\ (x < y)\ \texttt{then}\ \{x = x + y; y = y + 1; c = c + 1; d = d + 1\}\ \texttt{else}\ \{x = x + y; y = y + 1\}\},\ \texttt{IniVal} \rightarrow \{x = 0; y = 2; c = 1; d = 1\}]$$

$$\text{Output: } c = d \wedge y^2 = y + 2x + 2 \wedge (0 + 2 < 10 \Rightarrow x + y < 10 \vee (x = 9 \wedge y = 5))$$

Example 2.5. A relatively simple well-known example is taken from [7]. The invariant property returned by `Aligator` is given below.

$$\text{Input: } \texttt{Aligator}[\texttt{while}(x \neq y)\{x = x + 1; y = y - 1\}, \texttt{IniVal} \rightarrow \{x = a; y = b\}]$$

$$\text{Output: } x + y = a + b \wedge (a = b \Rightarrow x = a \wedge y = b) \wedge (a + b = 2x = 2y \vee a = b \vee x \neq y)$$

Internally, (i) `Aligator` first checks whether a given input is as (1). If it is not, an error is reported, and the invariant inference stops. Otherwise, polynomial equalities $p_i(X) = 0$ as invariants are generated for loop (1) with *omitted tests*, as follows: the closed form system of X is derived using recurrence solving over the loop body with the summation variable n, and variables depending on n are then eliminated by the Gröbner basis computation [8]. In the sequel, we write $CF_X(n)$ to mean the system of closed form expressions of X as functions of n. As a result of this step, valid polynomial relations among loop variables are inferred [16]. (ii) Next, the *loop condition is taken into account*, and it is checked whether conditional branches *commute*. By commuting we mean that variables $X' \subseteq X$ in the loop condition b_0 are changed by affine mappings in the same manner on each conditional branch. Deriving $CF_{X'}(n)$ is thus also feasible by means of recurrence solving in case of loops with nested conditionals, and, as discussed on page 335, $CF_{X'}(n)$ yields a *univariate polynomial system in* n. If

it is established that the branches commute, quantifier elimination is applied to derive bound assertions on n as additional polynomial inequalities $r_s(X) \geq 0$ and equalities $q_j(X) = 0$. This is based on a relatively simple idea: we seek solution to the formula given below, expressing the upper bound of n.

$$\exists n.n \geq 0 \wedge \Big(\big(\underbrace{b_0[\![n]\!] \wedge CF_{X'}(n)}_{L(n)} \big) \wedge \underbrace{\neg b_0[\![n+1]\!]}_{T(n)} \Big), \tag{2}$$

where $b_0[\![n]\!]$ represents the loop condition in which all variables X' have been substituted by their values $CF_{X'}(n)$ at iteration n; $L(n)$ encodes the behavior of X' during the nth loop iteration; and $T(n)$ corresponds to the termination criteria of the loop after n iterations. The formulas L and T are functions of the loop counter n and are derived from substituting variables X' by their closed forms $CF_{X'}(n)$. As mentioned on page 335, b_0 is expressed by linear inequalities over X', hence $b_0[\![n]\!]$ is expressed by polynomial inequalities in n. This way, solving (2) reduces to the problem of quantifier elimination from a quantified system of univariate polynomial inequalities and equalities in n, which can be solved as presented in [9]. Note that although [9] would also handle the case when the loop condition is a non-linear polynomial inequality, due to efficiency reasons, we only treat loops whose loop conditions are linear.

For solving (2) we rely on the quantifier elimination engine of `Mathematica` integrated into `Aligator` as part of the `AnalyseBound` command for inferring bound assertions. If an exact value of n can be computed, the values $V(X')$ of variables X' when exiting the loop are derived using $CF_{X'}(n)$. Note that equation (2) makes the assumption that the loop will be executed at least once. However, it is trivial to compute the symbolic state at the end of the loop when it is executed 0 times. Turning $V_{X'}$ into a loop invariant is based on the following simple fact. If the loop is executed at least once, then either the loop condition holds or values of the variables X' fulfill $V_{X'}$.

Finally, the loop invariant returned by `Aligator` is the conjunction of the polynomial invariants derived in step (i) and the invariants obtained after the bound analysis from step (ii).

Command 2.4 : `AnalyseBound[`$S_{X'}$`, `b_0`, IniVals]`

Input: Assignment statements $S_{X'}$ of loop (1) corresponding to the variables X', loop test b_0 and initial values of X'
Output: Formula $(b_0[\![0]\!] \Rightarrow b_0 \vee V(X')) \wedge (\neg b_0[\![0]\!] \Rightarrow CF_X(0))$

Example 2.6. For the loop of Example 2.1 we obtain:

$$\begin{aligned} &\texttt{Input:}\ \texttt{AnalyseBound}[\{cnt = cnt - 1\}, \\ &\qquad\qquad cnt > 0,\ \{res_0 = a, cnt_0 = b\}] \\ &\texttt{Output:}\ (b > 0 \Rightarrow cnt = 0 \vee cnt > 0) \wedge (b \leq 0 \Rightarrow res = a \wedge cnt = b) \end{aligned}$$

where we denote respectively by res_0 and cnt_0 the variables standing for the initial values of res and cnt. The polynomial invariant inferred in step (i) of `Aligator` is $res + cnt = a + b$. Hence, the complete invariant returned by `Aligator` to `VCG` is $cnt + res = a + b \wedge (b > 0 \Rightarrow cnt = 0 \vee cnt > 0) \wedge (b \leq 0 \Rightarrow res = a \wedge cnt = b)$, as also presented in Example 2.3.

Example 2.7. For the loop of Example 2.4 the result of `AnalyseBound` is:

$$\texttt{Input:}\ \texttt{AnalyseBound}[\{x = x + y; y = y + 1\}, \quad x + y < 10,\ \{x_0 = 0, y_0 = 2, c_0 = 1, d_0 = 1\}$$

$$\texttt{Output:}\ 0 + 2 < 10 \Rightarrow x + y < 10 \vee (x = 9 \wedge y = 5)$$

where, x_0, y_0, c_0 and d_0 denote respectively the initial values of x, y, c and d.

Note that the invariant generation in `Aligator` might perform unsafe optimizations (since data types are not yet handled). For instance, it always reduces $2(\frac{x}{2})$ to x, which is only valid when x is even. For this reason, in order to ensure correctness of invariants, and subsequently to ensure soundness of `Valigator`, generating and proving proof obligations for invariants' validity is crucial in `Valigator`.

2.3 Proving Verification Conditions

`VCG` generates proof obligations (verification conditions) in the SMT-LIB [6] format in one of the following two logics: Quantifier-free Bit Vectors (QF_BV) or Quantifier-free Linear Integer Arithmetic (QF_LIA). QF_LIA is more limited as it only supports linear relation among variables, but it is usually faster at proving or disproving obligations.

`Valigator` can either feed the proof obligations directly into a theorem prover or dump them to disk. `Valigator` has been tested with `CVC3` and `Z3` for linear arithmetic and `STP` for bit vectors. However, any other prover handling one of the mentioned logics and the SMT-LIB format could be used as well. Note that these two logics are decidable; the provers named above are sound and complete.

Example 2.8. Using `CVC3`, `Z3`, or `STP`, all proof obligations listed in Example 2.2 are successfully proved in less than 3 seconds on a 2.0GHz machine.

3 Conclusion

By combining automated reasoning and symbolic computation, `Valigator` allows us to verify programs annotated with pre- and postconditions, and offers automatic support for inferring invariant properties of P-solvable loops. The approach was successfully tested on several examples, some of which are presented in this paper.

So far, using only preconditions of loops, bound assertions are derived under the assumption that quantifier elimination yields an exact integer solution on the number of loop iterations. However, in many cases such a solution does not exist. A possible extension of our approach would be to use an interval approximation of the bound on the number of loop iterations.

The class of loops for which `Valigator` can automatically infer invariants is limited by constraints on the program structure; `Valigator` can handle branches inside the loop as long as they commute. We are trying to extend our approach to programs with non-commuting branches, and thus with more complex flow structure.

We are also interested in generalizing the framework to programs on non-numeric data structures such as arrays and lists.

References

1. Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., Schmitt, P.H.: The KeY Tool. Software and System Modeling 4(1), 32–54 (2005)
2. Barnes, J.: High Integrity Software - The Spark Approach to Safety and Security. Addison-Wesley, Reading (2003)
3. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: Proc. of FMC (2005)
4. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362. Springer, Heidelberg (2005)
5. Barrett, C., Berezin, S.: CVC Lite: A new implementation of the cooperating validity checker. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 515–518. Springer, Heidelberg (2004)
6. Barrett, C., Ranise, S., Stump, A., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB) (2008), http://www.SMT-LIB.org
7. Brauburger, J., Giesl, J.: Approximating the Domains of Functional and Imperative Programs. Sci. Comput. Programming 35(1), 113–136 (1999)
8. Buchberger, B.: An Algorithm for Finding the Basis Elements of the Residue Class Ring of a Zero Dimensional Polynomial Ideal. J. of Symbolic Computation 41(3-4), 475–511 (2006)
9. Collins, G.E.: Quantifier Elimination for the Elementary Theory of RealClosed Fields by Cylindrical Algebraic Decomposition. LNCS, vol. 33, pp. 134–183 (1975)
10. Cousot, P., Cousot, R.: Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: Proc. of POPL, pp. 238–252 (1977)
11. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
12. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a Theorem Prover for Program Checking. J. of the ACM 52(3), 365–473 (2005)
13. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, Englewood Cliffs (1976)
14. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended Static Checking for Java. In: Proc. of PLDI, pp. 234–245 (2002)
15. Ganesh, V., Dill, D.L.: A Decision Procedure for Bit-Vectors and Arrays. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590. Springer, Heidelberg (2007)
16. Kovács, L.: Reasoning Algebraically About P-Solvable Loops. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 249–264. Springer, Heidelberg (2008)
17. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary Design of JML: A Behavioral Interface Specification Language for Java. Technical Report 98-06u, Iowa State University (2003)
18. Müller, P., Meyer, J., Poetzsch-Heffter, A.: Programming and Interface Specification Language of Jive— specification and Design Rationale. Technical Report 223, University of Hagen (1997)
19. Nipkow, T., Paulson, L.C., Wenzel, M.T.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)

20. Odersky, M.: The Scala Language Specification (2008), http://www.scala-lang.org
21. Owre, S., Shankar, N., Rushby, J.: VS: A Prototype Verification System. In: Kapur, D. (ed.) CADE 1992. LNCS, vol. 607. Springer, Heidelberg (1992)
22. Seidl, H., Petter, M.: Inferring Polynomial Invariants with Polyinvar. In: Proc. of NSAD (2005)
23. van den Berg, J., Jacobs, B.: The LOOP Compiler for Java and JML. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 299–312. Springer, Heidelberg (2001)
24. Wolfram, S.: The Mathematica Book. Version 5.0. Wolfram Media (2003)

Reveal: A Formal Verification Tool for Verilog Designs

Zaher S. Andraus, Mark H. Liffiton, and Karem A. Sakallah

Department of Electrical Engineering and Computer Science
University of Michigan, Ann Arbor, MI 48109-2122
{zandrawi,liffiton,karem}@umich.edu

Abstract. We describe the Reveal formal functional verification system and its application to four representative hardware test cases. Reveal employs counterexample-guided abstraction refinement, or CEGAR, and is suitable for verifying the complex control logic of designs with wide datapaths. Reveal performs automatic datapath abstraction yielding an approximation of the original design with a much smaller state space. This approximation is subsequently used to verify the correctness of control logic interactions. If the approximation proves to be too coarse, it is automatically refined based on the spurious counterexample it generates. Such refinement can be viewed as a form of on-demand "learning" similar in spirit to conflict-based learning in modern Boolean satisfiability solvers. The abstraction/refinement process is iterated until the design is shown to be correct or an actual design error is reported. The Reveal system allows some user control over the abstraction and refinement steps. This paper examines the effect on Reveal's performance of the various available options for abstraction and refinement. Based on our initial experience with this system, we believe that automating the verification for a useful class of hardware designs is now quite feasible.

1 Introduction

The paradigm of iterative abstraction and refinement has gained momentum in recent years as a particularly effective approach for the scalable verification of complex hardware and software systems. Dubbed *counterexample-guided abstraction refinement* (CEGAR), its power stems from the elimination (i.e., abstraction) of details that are irrelevant to the property being checked and from analyzing any spurious counterexamples to pinpoint and add just those details that are needed to refine the abstraction, i.e., to make it more precise. Originally pioneered by Kurshan [13], it has since been adopted by several researchers as a powerful means for coping with verification complexity. In particular, the use of abstraction-based verification has been thoroughly studied in the context of model checking by Clarke *et al.* [6] and Cousot and Cousot [7] for over two decades. Later methods by Clarke *et al.* [6], Jain *et al.* [12] and Das *et al.* [8] have successfully demonstrated the automation of abstraction and refinement in the context of model checking for safety properties.

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 343–352, 2008.

Whereas such a verification paradigm is appealing at a conceptual level, its success in practice hinges on effective automation of the abstraction and refinement steps, as well as the various checking steps requiring sophisticated reasoning. In this paper, we describe how these issues are addressed by Reveal, an automatic CEGAR-based verification system. Reveal is used to formally verify complex hardware designs, including pipelined microprocessors whose RTL descriptions have tens of thousands of HDL source lines, thousands of signals, and hundreds of thousands of state bits.

Below, we will describe Reveal's CEGAR flow and analyze its behavior and performance by way of four representative test cases. For each test case, we compare a number of methods to model and check the desired properties on the abstract design, including the use of a *Satisfiability Modulo Theories* (SMT) solver [9]; we study trade-offs between various refinement options; we highlight the types of lemmas generated in the refinement stage and analyze the idiosyncrasies leading to them; we show how genuine bugs were discovered using Reveal; we provide experimental evidence that demonstrates the importance of datapath abstraction for the scalability of formal verification; and, finally, we compare the performance of Reveal against a number of existing automatic tools that perform formal verification of hardware, such as VCEGAR [12], BAT [14], UCLID [4], and VIS [10].

The rest of the paper is organized in four sections. Section 2 reviews Reveal's CEGAR framework, and Sections 3 and 4 describe our benchmark test cases and how they were verified using the Reveal system. Finally 5 summarizes the paper's conclusions.

2 The Reveal Verification System

Figure 1 depicts the flowchart of the reveal system. Reveal performs checks of safety properties on hardware designs described in the Verilog hardware description language (HDL). A typical usage scenario involves providing two Verilog descriptions of the same hardware design, such as a high-level specification and a detailed implementation, and checking them for functional equivalence. Reveal adopts the CEGAR-based approach of Andraus *et al.* [1], mainly involving:

Abstraction. The goal of abstraction is to obtain a compact representation of the design for which formal property checking is more likely to terminate (i.e., to scale both in time and space) than if applied directly on the original design. Reveal performs datapath and memory abstraction by replacing datapath units with *uninterpreted functions* and *predicates*, while leaving the control logic unabstracted. This allows reasoning about the complex control of the design, while avoiding the complexity introduced by datapath elements. Previous work (e.g. [5]) showed that it is possible to prove many useful (equivalence) properties if the abstract model is expressed in the logic of Equality with Uninterpreted Functions (EUF), a quantifier-free fragment of first order logic. Scalability can be further improved by abstracting to CLU, which extends EUF with counting arithmetic and lambda expressions for memories [4].

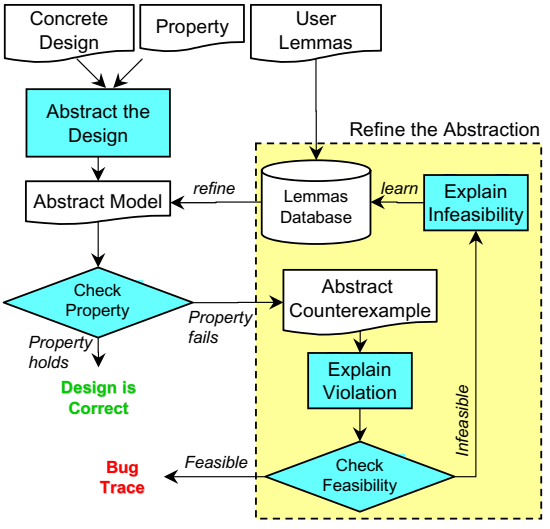


Fig. 1. The Reveal Flow

Table 1. Benchmark Statistics

Name	Verilog Lines	Verilog Signals	State Bits
Sorter	79	30	35 to 1027
DLX	$2.4x10^3$	399	$1.0x10^{11}$
Risc16F84	$1.2x10^3$	169	$1.0x10^5$
X86	$1.3x10^4$	$1.0x10^3$	$5.8x10^3$

Property Checking. Formal reasoning in the EUF or CLU logics determines if the abstracted design satisfies the specified property. Early EUF/CLU solvers convert the formula to an equi-satisfiable propositional formula and use an off-the-shelf Boolean solver to check for satisfiability. In contrast, Satisfiability Module Theories (SMT) solvers (e.g. YICES [9]) operate on these formulas directly by integrating specialized theory solvers within a backtrack propositional solver. SMT solvers, thus, are able to take advantage of the high-level semantics of the non-propositional constraints, while at the same time benefiting from the powerful reasoning capabilities of modern propositional SAT solvers. Reveal uses the YICES solver [9] [16] in the property checking step, allowing the integration of the *empty theory* (consistency of term equality), *UF theory*, and *integer theories* (for counting).

Refinement. An abstract counterexample, demonstrating that the property is violated by the abstract model, has to be checked for feasibility on the concrete model. If feasible, a concrete counterexample trace is generated. If not, the counterexample is spurious, and is *refuted* in the abstract model by adding a blocking clause, similar to learning in SAT, and the process iterates. Reveal avoids the naïve refutation of one counterexample at a time, which usually leads to very slow convergence, rendering the approach impractical. Instead, one or more succinct explanations are used in each iteration to explain infeasibility and refine the abstraction for the next round of checking. These explanations, referred to as *lemmas*, are universal facts extracted from the concrete model to refute current or future spurious counterexamples and can thus be stored in a lemma database and re-used across invocations of Reveal on the same (family of) design(s). Preliminary results of this scheme [3] show that the convergence of the refinement

loop is contingent upon the way these lemmas are derived. Reveal employs a reasoning engine that combines YICES with CAMUS [11] during feasibility and refinement. The CAMUS tool can derive one, multiple, or all minimally unsatisfiable subsets (MUSes for short) of constraints from a set of infeasible constraints. Finding MUSes allows for trimming the infeasibility explanations, effectively enlarging their footprint in the abstract solution space. Finding multiple MUSes means learning multiple lemmas in each refinement iteration and yielding a significant speedup in the convergence of the refinement loop. Finally, scalability is further improved by finding MUSes with the use of the *bit-vector theory* in YICES. Earlier methods (e.g. [3]) use a bit-blasting approach in which the abstract counterexample is encoded with propositional constraints and passed to a propositional solver. In contrast, reasoning at the word-level with the bit-vector theory in YICES allows much more efficient derivation of MUSes (i.e., lemmas).

3 Case Studies

We performed our experiments on the four designs which we briefly describe in this section. Table 1 summarizes the design statistics. Interested readers can find more details in [2].

Sorter Case Study. The Sorter design implements two versions of an algorithm that sorts four bit-vectors. The computation delay in both versions is 3 cycles. The property we verified is the equality between corresponding outputs in the two versions. All the bit-vectors in the two units, including the inputs and the outputs, have bit-width W, which we vary from 2 to 64 to see the effect of the datapath width on the scalability of each tool. Reveal's performance on the Sorter is presented in Figure 2, and will be analyzed in Section 4.

DLX Case Study. DLX is a 32-bit RISC microprocessor [11]. Its salient features include a 32-bit address space with separate instruction and data memories, a 32-word register file with two read ports and one write port, and 38 op-codes for arithmetic, logical, and control operations. Our case study involved proving the equivalence of two versions of DLX [17]. The first version, which we will refer to as *DLXSpec*, is a single-cycle implementation of the instruction set architecture (ISA) and serves as the architectural specification of the microprocessor. The second version, labeled *DLXImpl*, is a standard 5-stage pipeline.

RISC16F84 Case Study. This design is an implementation of the Risc16F84 microcontroller [19]. It has a 213x14-bit instruction memory, a 29x8-bit data memory, 34 op-codes, and a 4-stage pipeline. Similarly to the DLX case, we denote the implementation and specification by *OCImpl* and *OCSpec* respectively. *OCImpl* processes one instruction every four cycles, while *OCSpec* needs one cycle to process each instruction.

X86 Case Study. The X86 design is an open source RTL Verilog model developed at IIT Madras that implements Intel's IA-32 ISA [18]. The design's

Decoder module is responsible for fetching an instruction prefix from memory, finding the total length of the instruction, and fetching and decoding the rest of the instruction. We verified the property that the Decoder activates the corresponding decode unit (Integer versus Floating Point) when the instruction is confined to a set of 6 integer and floating point op-codes.

4 Results and Analysis

We verified a number of buggy and bug-free variations of each of the aforementioned designs. The buggy versions were obtained by injecting errors in the RTL description. These variations are described in Table 2. Columns labeled T, I, and L, describe, respectively, total run-time (seconds), number of iterations, and total number of refinement lemmas (when applicable). 'TO' stands for "Time Out" (600s). Finally, the smallest run-time is highlighted in bold in each row; there can be multiple in each row when the difference is insignificant.

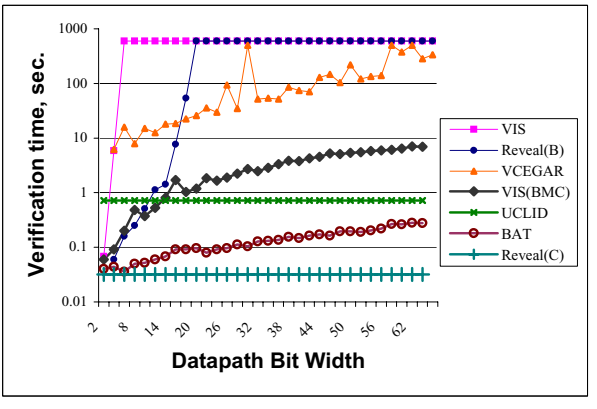


Fig. 2. Runtime Graphs for Sorter

```
// DLX
define BEQ 4
define op 31:26
initial OR3 = 32'd0
case IR3[`op] `BEQ: ...

// X86
op2 = 32'd0;
if (...) op2[16:0] =
    instrSeq[31:16];
```

Fig. 3. Verilog Code Fragment from DLX and X86

Reveal's modes were classified by a one-, two-, or three-letter code that indicates the abstraction and refinement options used. Abstraction options are labeled B (bit-level, i.e., no abstraction), C (CLU abstraction), and E (EUF abstraction). Refinement options are labeled V (refinement via refuting the abstract violation) and L (refinement with lemmas). For lemma refinement, S denotes refinement with a single lemma per iteration, while M denotes refinement with multiple lemmas. For example, the label CLM means CLU abstraction and refinement with multiple lemmas, whereas EV means EUF abstraction and refinement with the negation of the abstract violation.

Our empirical case study compares the performance of Reveal against the verification systems UCLID, BAT, VCEGAR, and VIS on a 2.2 GHz AMD Opteron processor with 8GB of RAM running Linux. UCLID [4] allows modeling of the datapath with abstract terms, and memories with Lambda expressions. BAT [14] models memories with set and get functions for reads and writes, respectively, but models the datapath with finite-length bit-vectors. VCEGAR [12] performs word-level predicate abstraction on the Verilog input but does not abstract memory arrays. Finally, VIS [10] uses, by default, bit-level reachability analysis to verify invariants. It can also be used in two special modes: one that performs bounded model checking of safety properties (denoted herein by VIS(BMC)), and another that performs invariant checking with a CEGAR algorithm based on hiding registers [15] (denoted by VIS(AR)).

#	Test Case/Version	CV		ELS		CLS		ELM			CLM			B
		T	I	T	I	T	I	T	I	L	T	I	L	T
D1	Bug-free DLX	TO	>1507	1.92	9	1.8	8	0.6	4	8	1.0	6	12	TO
D2	Pipeline "Stall" s-a-1	0.11	1	0.15	1	0.12	1	0.11	1	0	0.1	1	0	0.21
D3	Incorrect 'jump' address	3.16	45	2.22	11	1.16	5	1.13	3	5	1.1	4	8	6.7
R1	Bug-free RISC16F84	TO	>1767	TO	>1204	TO	>1085	257	93	185	148	68	170	209
R2	Floating "carry-in" signal	0.79	8	56	20	TO	>1881	72	44	13	40	33	39	15.2
R3	"aluout_zero_node" s-a-1	115	654	50	123	121	311	2.6	5	15	27.3	40	73	11.6
X1	Bug-free X86	TO	>388	TO	>1158	TO	>945	36.5	40	104	60.4	19	96	TO
X2	en_{Int} and en_{FP} swapped	TO	>461	TO	>1062	TO	>1046	30.5	78	161	103	24	86	TO
X3	Wrong FSM transitions	1.98	2	1.95	2	1.96	2	2.0	2	6	2.1	1	0	2.72
X4	CMP activates the FP unit	TO	>308	TO	>847	TO	>1252	23	12	41	58.7	7	43	TO

Fig. 4. Verification results for the DLX, RISC16F84, and X86 variations

4.1 Datapath Abstraction

The merits of datapath abstraction are evident in all verification runs. In particular, the performance of Reveal(C) and UCLID on the Sorter example is oblivious to the datapath bit-width W (Figure 2). In both cases, the abstract model is unaltered when the datapath bit width is changed; thus, the time needed to verify the abstract model is constant. Furthermore, the only interaction between the datapath and the control logic in this design involves bit-vector inequalities, allowing the CLU logic to prove the property without any refinement. The performance of the remaining tools degrades as W increases:

- VCEGAR takes 6.1 seconds to prove the property for W=2 as it incrementally discovers between 33 and 40 predicates within 58 to 130 iterations. Additionally, run-time grows exponentially with W. We suspect that the reason behind this is the expense of simulating the abstract counterexample on the concrete design in each refinement iteration, as well as the repeated generation of the abstract model each time a new predicate is added.
- The run-times of Reveal(B), VIS, and VIS(BMC) degrade rapidly as the bit width is increased. The run-times of VIS(AR) are similar to those of VIS and were removed from the graph to avoid clutter.

– BAT's performance degrades with increasing W, but BAT's reduction of the verification formulas to CNF appears to play an important role in keeping the run-time low.

Note that Reveal(B) (in Table 2) has the worst performance, though surprisingly it is able to terminate on a number of buggy versions. This is attributed to the ability of the bit-vector solver in YICES to efficiently find a satisfying assignment and thus its ability to find abstract counterexamples. However, the rest of the cases confirm that proving that a property *holds* is intractable without abstraction.

Finally, comparing Reveal(C) and Reveal(E) sheds some light on the difference between abstraction to EUF or CLU. In particular, Reveal(C) converges faster than Reveal(E) in terms of refinement iterations in the X86 and RISC16F84 cases. This is attributed to the heavy use of counters in these designs. Still, Reveal(E) outperforms Reveal(C) in most cases since the latter uses an integer solver which impacts overall performance.

4.2 Refinement Convergence

The performance of the various options in Reveal demonstrate the role of automatic refinement. In particular, Table 2 shows that the use of lemmas for refinement (modes ELS, CLS, ELM, and CLM) is far superior to refuting one counterexample at a time (mode CV). Also, using multiple lemmas in each refinement iteration (modes CLM and ELM) outperforms refinement with a single lemma at a time (modes ELS and CLS). The R2 case shows an interesting outlier; Reveal(CV) is significantly faster than any version that refines with lemmas. This is due to the heuristic nature of the satisfiability search for finding a bug. Any search, regardless of the refinement used, could "get lucky" and find a bug early, though only rarely.

To further assess the effect of lemmas on the convergence of the algorithm, we ran Reveal(C) on a version that combines the three bugs present in X2, X3, and X4. This was an iterative session, in which Reveal was re-invoked after correcting each reported bug. We tested Reveal in two modes: a mode in which learned lemmas are discarded after each run and a mode in which learned lemmas are saved and used across runs. The total run-time for the first mode was 232 seconds, whereas the run-time in the second mode was 166 seconds, a 40% improvement in speed. This confirmed our conjecture that lemmas discovered in one verification run can be profitably used in subsequent runs. The verification of real-life designs involves tens to hundreds of invocations of the tool, thus a significantly larger speed-up could be seen in practice.

4.3 Refinement Lemmas

We traced the source of refinement lemmas back to the original Verilog code involving control/datapath interactions. For example, most of the lemmas in the DLX example were related to the pipeline registers and control logic in *DLXImpl*,

such as the lemma $(IR3 = 32'd0) \rightarrow (IR3\,[31:26]) \neq 6'd4$, which states that it's not possible to extract a non-zero field from a zero bit-vector. The source of the lemma is in Figure 3 and it involves IR3; the initial abstraction lost the fact that IR3[31:26] can not be equal to 4, and it found a spurious counterexample that executed the BEQ instruction. Another example is the set of lemmas in the RISC16F84, most of which are due to the *variable opcode width* feature, wherein the opcode field can be k-bits wide for any $k \in K = \{2, ..., 7, 14\}$. For instance, the opcode of the *goto* instruction is IR[13:11]=3'b101, while the opcode for *addlw* is IR[13:9]=5'b11111. The encoding guarantees that only one opcode is active at any given time. This information is lost when abstracting the bit-vector extraction operation. This results in the occurrence of lemmas of the form $(IR\,[13:k_1] = v_1) \rightarrow (IR\,[13:k_2] \neq v_2)$ for values $v_1 \neq v_2$ and distinct indices $k_1, k_2 \in K$.

It is worth mentioning that our experience with this flow shows that refinement lemmas can be very simple, or very complex, depending on the design and the property. In either case, the automatic discovery and (on-demand) refinement of only those relevant ones is an important enabler for the scalability of this approach.

4.4 Discovering Genuine Bugs

In addition to discovering artificially introduced bugs (e.g. those described in Table 2), Reveal was able to discover a number of *genuine* bugs. In particular, the RISC16F84 design includes the Verilog expression {1'b0,aluinp2_reg,c_in} in *OCImpl*, which uses a *floating* signal c_in as the carry-in bit to an 8-bit adder. In contrast, *OCSpec* performs addition without any carry-in bit. Reveal thus produces a counterexample showing the deviation with c_in assigned to 1. The unit designer acknowledged this problem and asserted that the simulation carried out for this design assumed c_in=0. An additional coding problem was discovered in X86; the RTL description includes the code given in Figure 3, which extracts a 16-bit displacement value from the instruction stream and assigns it to a 17-bit register. Most synthesis tools will zero-extend the RHS expression to make the sizes consistent, in which case the resulting model is still correct. Nonetheless, such an error may indicate additional problems in other units of the design.

4.5 Performance of VIS, VCEGAR, and UCLID

VIS, VCEGAR, and UCLID were not able to successfully terminate on the DLX, RISC16F84, or X86 designs. In some cases, the tool times out or exceeds available memory, and in others, an internal error causes unexpected termination. Details of these experiments can be found in [2].

5 Conclusions

We examined the performance of Reveal, a CEGAR-based formal verification system for safety properties in general, and equivalence in particular. Reveal is

particularly suited for the verification of designs with wide datapaths and complex control logic. Datapath abstraction allows Reveal to focus on the control interactions making it possible to scale up to much larger designs than is possible if verification is carried out at the bit level. Additionally, Reveal's demand-based lemma generation capability eliminates one of the obstacles that had complicated the deployment of formal equivalence tools in the past. From a practical perspective, hands-free operation and support of Verilog allow Reveal to be directly used by designers. These capabilities were demonstrated by efficiently proving the existence of bugs, or proving the lack thereof, in four Verilog examples that emulate real-life designs both in terms of size and complexity.

Acknowledgements

This work was funded in part by the DARPA/MARCO Gigascale Systems Research Center, and in part by the National Science Foundation under ITR grant No. 0205288.

References

1. Andraus, Z., Liffiton, M., Sakallah, K.: Refinement strategies for verification methods based on datapath abstraction. In: Proc. of Asia and South Pacific Design Automation Conference, pp. 19–24 (2006)
2. Andraus, Z., Liffiton, M., Sakallah, K.: CEGAR-based formal hardware verification: a case study. Technical Report CSE-TR-531-07, University of Michigan (2007)
3. Andraus, Z., Sakallah, K.: Automatic abstraction and verification of Verilog models. In: Proc. of Design Automation Conference, pp. 218–223 (2004)
4. Bryant, R., Lahiri, S., Seshia, S.: Modeling and verifying systems using a logic of counter arithmetic with Lambda expressions and uninterpreted functions. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 78–92. Springer, Heidelberg (2002)
5. Burch, J., Dill, D.: Automatic verification of pipelined microprocessor control. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 68–80. Springer, Heidelberg (1994)
6. Clarke, E., Grumberg, O., Long, D.: Model checking and abstraction. ACM Transactions on Programming Languages and Systems (TOPLAS) 16(5), 1512–1542 (1994)
7. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conference Record of the Sixth Annual ACM SI PLAN-SIGACT Symposium on Principles of Programming Languages, pp. 238–252 (2006)
8. Das, S., Dill, D.: Successive approximation of abstract transition relations. In: IEEE Symposium on Logic in Computer Science, pp. 51–58 (2001)
9. Dutertre, B., de Moura, L.: A fast linear arithmetic solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)
10. Brayton, R., et al.: VIS: a system for verfication and synthesis. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 428–432. Springer, Heidelberg (1996)

11. Hensessy, J., Patterson, D.: Computer Architecture: A Quantitative Approach, 2nd edn. Morgan Kaufmann, San Francisco (1996)
12. Jain, H., Kroening, D., Sharygina, N., Clarke, E.: Word-level predicate abstraction and refinement for verifying RTL Verilog. In: Proc. of Design Automation Conference, pp. 445–450 (2005)
13. Kurshan, R.: Computer-Aided Verification of Coordinating Processes: The Automata-Theoritic Approach. Princeton University Press, Princeton (1999)
14. Manolios, P., Srinivasan, S., Vroon, D.: Automatic memory reductions for rtl model verification. In: Proc. of Int'l. Conference on Computer-Aided Design, pp. 786–793 (2006)
15. Wang, F., Li, B., Jin, H., Hachtel, G., Somenzi, F.: Improving Ariadne's Bundle by following multiple threads in abstraction refinement. In: Proc. of Int'l. Conference on Computer-Aided Design, pp. 408–415 (2003)
16. http://yices.csl.sri.com/
17. http://www.eecs.umich.edu/vips/stresstest.html
18. http://vlsi.cs.iitm.ernet.in/x86_proj/x86Homepage.html
19. http://www.opencores.org

A Formal Language for Cryptographic Pseudocode

Michael Backes[1,2], Matthias Berg[1], and Dominique Unruh[1]

[1] Saarland University, Saarbrücken, Germany
[2] MPI-SWS

Abstract. Game-based cryptographic proofs are typically expressed using pseudocode, which lacks a formal semantics. This can lead to ambiguous specifications, hidden mistakes, and even wrong proofs. We propose a language for expressing proofs that is expressive enough to specify all constructs occurring in cryptographic games, including probabilistic behaviors, the usage of oracles, and polynomial-time programs. The language is a probabilistic higher-order lambda calculus with recursive types, references, and support for events, and is simple enough that researchers without a strong background in the theory of programming languages can understand it. The language has been implemented in the proof assistant Isabelle/HOL.

1 Introduction

A salient technique for formulating cryptographic security proofs are so-called sequences of games. In such a proof, a desired security property is first formulated as a probabilistic experiment, the initial *game*.[1] Usually, this game comprises the adversary as well as the protocol under consideration and describes the actions that the adversary may perform. The output of the game denotes if the adversary performs a successful attack (where the notion of a successful attack depends on the security notion under consideration).

To find the probability of an attack, the game is then transformed in a series of steps, and for each step it is shown that the new game is in some way similar to the previous one. E.g., it may be shown that the transformation changes the probability of an event at most by a small amount ε. The last game in the sequence usually has a trivial form in which the probability of an attack can be directly bounded using a complexity-theoretic assumption. We refer to [32,7] for a thorough overview of this technique.

This game-based technique leads to well-structured proofs. Ideally, each of the transformations can be verified individually without having to reason about other transformations. This simplifies detecting potential mistakes in the proof and where these mistakes occur. Further, since the individual transformations are typically very simple (there is a trade-off between the complexity of the

[1] Games in the sense of this technique must not be confused with game-theoretic games. In our context a game is just the description of some probabilistic process.

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 353–376, 2008.

transformations and the length of the overall game sequence), and since the correctness of each transformation can be proven independently, the game-based proof technique should constitute an ideal candidate for formal verification of complexity-based cryptographic proofs.

In addition to structuring for cryptographic proofs, games are also the most popular method for formulating security properties in cryptography. This combines nicely with the game-based proof technique as the security definition already describes the initial game of the game sequence. Thus no error-prone switch of paradigm is needed at the beginning of a game-based proof.

In practice, however, the advantages of game-based proofs are mitigated by the following limitation: in virtually all cryptographic publications, the games are either described in words, or—at best—in some ad-hoc pseudo-code. In both cases, no formal semantics of the language used are specified. This leads to three grave disadvantages:

- *Ambiguity of definitions.* As cryptographic games are used to provide security definitions, the lack of a formal semantics may result in different interpretations of the details of a given game, and hence in an ambiguous security definition. For example, if a subroutine representing an adversary is invoked twice, it might be unclear whether the adversary may keep state between these two invocations. The author of the security definition may explicitly point out these ambiguities and resolve them; this, however, assumes that the author is aware of all possible interpretations.
- *Mistakes may be hidden.* The correctness of a cryptographic proof may be much harder to verify: Since the correctness of a transformation usually depends on the precise definition of the game, the reader of the proof may not be sure whether the transformation is indeed incorrect or whether the reader just misinterpreted the definition of the game. Moreover, it may happen that for a sequence of three games A, B, C, depending of the exact definition of B, either the transformation from A to B, or that from B to C is incorrect. However, if the transformations are verified individually, in each case there is some interpretation of the meaning of B that lets the corresponding proof step seem correct.
- *Unsuited for machine-assisted verification.* Finally, if we are interested in machine-assisted verification of cryptographic protocols, the semantics of the games need to be defined precisely since a computer will not be able to reason about a semantics-free pseudo-code.

1.1 Our Contribution

We overcome these limitations by defining a language for formalizing cryptographic games. Our language has the following properties:

- *Expressiveness.* The language should be able to express all constructs that usually occur in the specification of cryptographic games, including probabilistic behaviors, the usage of oracles, and potentially continuous probability

measures for reasoning about, e.g., an infinitely long random tape for establishing information-theoretic security guarantees. From a language perspective, oracles are the higher-order arguments that are passed to a program. We consequently propose a higher-order functional probabilistic language to deal with sophisticated objects such as oracles (functional since functional languages deal with higher-order objects much more naturally than imperative languages). A purely functional language, however, is insufficient, because functional reasoning would not allow oracles to keep state between their invocations (and passing on state as explicit inputs would result in secrecy violations, e.g., an oracle would output its secret key to the adversary). We remedy this problem by including ML-style references in the language. Finally, events constitute a common technique in game-based cryptographic proofs for identifying undesirable behavior. We thus explicitly include events in the language. The semantics is operational in order to be able to introduce the notion of polynomial runtime.

– *Simplicity.* The definition of the language should be as simple as possible so that researchers without a strong background in the theory of programming languages can understand at least its intuitive meaning. In particular, the language should also have a syntax that is readable without a detailed introduction into the language.
– *Mechanization.* Our language has been implemented in the proof assistant Isabelle/HOL so that it is possible to use the power of its logic to reason about the language and to formally verify proofs. Moreover, we formalize several common relations between games in Isabelle/HOL, such as denotational equivalence, observational equivalence, and computational indistinguishability. We expect our language and these game transformations to culminate in a tool that enables cryptographers to check the validity of their proofs in a machine-assisted manner without having to bother about the details and intricacies of our language.

1.2 Related Work

The analysis of cryptographic protocols using formal techniques has received considerable attention [25,23,17,21,29,33,1,6]. These approaches, however, only provide guarantees with respect to a symbolic model that abstracts from the actual cryptography by means of a term algebra, so-called Dolev-Yao models. While there has been a substantial amount of work on carrying these results over to the computational model [2,18,4,19,24,3,12,10], these works only apply to standard cryptographic operations such as encryption schemes and digital signatures. They do, however, support concise reasoning about low-level crypto proofs, which is the goal of our work.

The work that comes closest to our framework is CertiCrypt [5], which is a framework for reasoning about game-based cryptographic proofs in the Coq proof assistant [34]. Their language takes into account complexity-theoretic issues of programs, permits using variables, and has a library for expressing common

security properties such as IND-CPA for encryption schemes as well as computational hypotheses such as the Decisional Diffie-Hellman (DDH) assumption. In contrast to our work, their language, however, is not higher-order, and it is restricted to discrete probability measures; therefore, reasoning about oracles and information-theoretic security guarantees is infeasible. These two shortcomings apply to the work discussed below as well.

CryptoVerif [8] is a tool to support game-based cryptographic proofs. The unforgeability of the FDH signature scheme was asserted by the tool assuming the existence of one-way permutations [9]. CryptoVerif differs substantially from our framework in that it aims at generating the sequence of games itself, based on a collection of predefined transformations, while we aim at checking if a sequence of games provided by the user is computationally sound. These approaches should thus be considered complementary instead of competing.

Other approaches for reasoning about game-based cryptographic proofs have been presented in [11,28], where they were used to establish the security of the ElGamal encryption scheme provided that the DDH assumption holds. The formalism in [11] is not powerful enough to fully express the security properties of interest, e.g., notions such as negligible probability or polynomial-time adversaries cannot be expressed. The formalism in [28] models adversaries immediately as Coq functions. The framework is hence only capable of offering limited support for proof automation because there is no special syntax for writing games that could be conveniently used to mechanize syntactic transformations. Moreover, the formalism ignores complexity-theoretic issues such as defining a polynomial-time adversary.

The creation of an automated tool to help with the mundane parts in proofs is advocated in [15]. The language presented in this paper is a step towards the completion of this goal.

1.3 Paper Outline

The paper is organized as follows. Section 2 introduces the notation that we will use throughout the paper. Section 3 provides a primer in probability theory. Section 4 defines the syntax of the language and its semantics. Polynomial runtime is formalized in Section 5. Section 6 defines relations over programs such as computational indistinguishability. Section 7 presents some fundamental tools for reasoning about our language. We show how to embed the language in higher-order logic in Section 8, and we provide examples of common cryptographic constructs expressed in our language in Section 9. Section 10 concludes and outlines future work.

2 Notation

Let $\mathbb{N}$ be the natural numbers including 0, $\mathbb{R}$ the real numbers, $\mathbb{R}^+$ the nonnegative real numbers including 0. Let $\mathbb{B}$ denote the set of Booleans. The powerset of X is denoted by 2^X. $X \times Y$ is the Cartesian product of X and Y, and $X \to Y$ is

the function space from X to Y. For a set Y, we write $f^{-1}(Y)$ for $\{x.f(x) \in Y\}$. By $\lambda x.p$ we denote the function mapping a to p' where p' results from replacing x by a in p. (E.g., $\lambda x.x + x$ maps a to $2a$.) We write $[a_1, \ldots, a_n]$ for lists (finite sequences) and $[]$ for the empty list. Given lists a, b, by $a@b$ we denote the concatenation of a and b. By $x :: a$ we denote the list a with the element x prepended.

We further summarize the notation introduced in Section 3 below: We write Σ_X for the canonical σ-algebra over X, $f(\mu)$ for applying f to a distribution μ, and δ_x for the probability distribution that assigns probability 1 to x. $f \circ g$ denotes the composition of two kernels, and $\forall x \in \mu.\ P(x)$ denotes the fact that $P(x)$ holds almost surely with respect to the distribution μ. Given a distribution μ and a measurable set E, by $\mu|_E$ we denote the distribution that sets the probability of all events outside E to 0.

3 A Primer in Probability Theory

We give a compact overview of measures and probability theory in this section. For a detailed overview, we refer to a standard textbook on probability theory, e.g., [16]. A *σ-algebra* over a set X is a set $\Sigma_X \subseteq 2^X$ such that $\emptyset \in \Sigma_X$, and $A \in \Sigma_X \Rightarrow (X \setminus A) \in \Sigma_X$, and for any sequence $(A_i)_{i\in\mathbb{N}}$, $A_i \in \Sigma_X$ we have that $\bigcap_i A_i \in \Sigma_X$. The smallest σ-algebra containing all $A \in G$ for some set G is called the σ-algebra generated by G. We assume that there is some canonical σ-algebra Σ_X associated with each set X; we call the sets $A \in \Sigma_X$ *measurable sets*. We assume that $\Sigma_{X\times Y}$ is generated by all $A \times B$ with $A \in \Sigma_X$, $B \in \Sigma_Y$. $\Sigma_{X\to Y}$ is generated by all sets $\{f.f(x) \in A\}$ for $x \in X$, $A \in \Sigma_Y$. For countable X, $\Sigma_X = 2^X$. $\Sigma_{\mathbb{R}}$ is generated by all sets $[a, b]$ with $a, b \in \mathbb{R}$ (Borel-algebra); $\Sigma_{\mathbb{R}^+}$ is defined analogously.

A *measure* over X is a function $\mu : \Sigma_X \to \mathbb{R}^+$ with the following properties: $\mu(\emptyset) = 0$ and for a pairwise disjoint sequence $(A_i)_{i\in\mathbb{N}}$, $A_i \in \Sigma_X$, we have $\mu(\bigcup_i A_i) = \sum_i \mu(A_i)$ (countable additivity). We call μ a *probability measure* if $\mu(X) = 1$ and we call it a *subprobability measure* if $\mu(X) \leq 1$. Intuitively, $\mu(E)$ denotes the probability that some value $x \in E$ is chosen. A subprobability measure can be used to model the output of a program that does not terminate with probability 1, then $\mu(X)$ is the probability of termination. The words measure and *distributions* are synonyms. We write δ_x for the *Dirac measure*, i.e., $\delta_x(E) = 1$ if $x \in E$, and $\delta_x(E) = 0$ otherwise. Given a measure μ on X and a set $E \in \Sigma_X$, we define the restriction of μ to E as $\mu|_E(A) := \mu(A \cap E)$. Intuitively, $\mu|_E$ sets the probability of all events outside E to 0.

A function $X \to Y$ is called *measurable* if for all $E \in \Sigma^Y$, we have $f^{-1}(E) \in \Sigma^X$. We can apply a measurable function f to a measure μ by defining $\mu' := f(\mu)$ by $\mu'(E) := \mu(f^{-1}(E))$. Intuitively, $f(\mu)$ is the distribution of $f(x)$ when x is chosen according to μ.

A function $f : X \to (\Sigma_Y \to \mathbb{R}^+)$ is called a *kernel* if for all $x \in X$ we have that $f(x)$ is a measure and if for all $E \in \Sigma_Y$ we have that the function $f_E : X \to \mathbb{R}^+$, $f_E(x) := f(x)(E)$ is measurable. Intuitively, a kernel is a *probabilistic function*

from X to Y that assigns every input $x \in X$ a distribution over outputs in Y. We call f a Markov kernel, if $f(x)$ is a probability measure for all $x \in X$, and we call f a *submarkov kernel* if $f(x)$ is a *subprobability* measure for all $x \in X$.

For any measurable function $f : X \to \mathbb{R}$ and any distribution μ over X, we can define the *(measure) integral* $\int f(x)\, d\mu(x)$. Intuitively, $\int f(x)\, d\mu(x)$ is the expected value of $f(x)$ if x is chosen according to the distribution μ. We refer to [16] for the formal definition of the integral. If f is bounded from below and above, then the integral always exists (and is finite).

Given a kernel g from Y to Z and a measure μ on Y, we define the *application* $g \cdot \mu$ by $(g \cdot \mu)(E) := \int g(w)(E) d\mu(w)$. Intuitively, $g \cdot \mu$ is the distribution resulting from choosing x according to μ and then applying the probabilistic function g. Given a kernel f from X to Y, we can define the *(kernel) composition* $g \circ f$ by $(g \circ f)(x) := g \cdot (f(x))$.

If μ is a measure over X and $P : X \to \mathbb{B}$ is a measurable function, we write $\forall x \leftarrow \mu. P(x)$ to denote $\exists A \in \Sigma_X. \mu(A) = 0 \wedge \forall x \in (X \setminus A). P(x)$. Intuitively, this means a value x sampled according to μ always satisfies $P(x)$.

4 Syntax and Semantics of the Language

This section introduces the syntax and the semantics of our language. We start by outlining the cryptographic requirements the language should fulfill, proceed by giving the syntax of the language and conclude by giving its operational semantics.

4.1 Cryptographic Requirements of the Language

We strive for defining a language that is powerful enough to express and reason about the constructions and the definitions that are used in cryptography. Since cryptography heavily relies on the use of probabilism, the language necessarily needs to be probabilistic. Moreover, the language should not be restricted to discrete probability measures, since discrete probability measures are not sufficient for reasoning about infinite constructions such as the random selection of an infinitely long random tape, e.g., to reason about information-theoretic security guarantees. Oracles, i.e., objects that can be queried in order to perform tasks such as computing the hash of some string or computing the decryption of some ciphertext, constitute another salient concept in modern cryptography. From a language perspective, oracles are higher-order arguments that are passed to a program. We hence strive for a higher-order functional language – higher-order to deal with sophisticated objects such as oracles, functional since functional languages deal with higher-order objects more naturally than imperative languages.

A purely functional language, however, is insufficient, because functional reasoning would, e.g., not allow oracles to keep state between its invocations. While one can rewrite every program that uses state into a purely functional, equivalent program without state by passing state around as an explicit object, this approach is insufficient as well in our setting because it inadequately deals

with secrecy properties: Consider an adversary that accesses a decryption oracle which, upon its first query, generates a secret key and subsequently decrypts the queries obtained from the adversary using that key. A purely functional setting with an explicit encoding of state would cause the oracle to return its state, i.e., its secret key, and the adversary had to additionally provide the key in its next query to the oracle. Clearly, this violates the intended secrecy of the key, and there is moreover not even a guarantee that the adversary provides the same, correct state in all of its queries. We remedy this problem by including ML-style references in the language.

To make the language efficiently usable from a programmer's perspective, we need a way to express data structures like lists and trees. Because we do not want to commit ourselves to specific data structures, we include mechanisms that are sufficient for the programmer to define his own type constructors. More precisely, the language has an iso-recursive type system [30] that includes product and sum types. This is enough to express arbitrary data types.

The use of events is a common technique in game-based cryptographic proofs. A game raises an event whenever some – usually unwanted – condition occurs. One then transforms this game into another game that behaves identical to the original game, except that it may proceed arbitrarily once the event is raised. Finally, one exploits the fundamental lemma of game-based cryptographic proofs, which shows that the error this transformation introduces is bounded by the probability that the event occurs. For conveniently reflecting this reasoning, we explicitly include events in the language.

4.2 Syntax of the Language

We now introduce the syntax of our language. Following the considerations of the previous section, the syntax relies on a probabilistic higher-order lambda calculus with references, iso-recursive types, and events. We first give the syntax in the following definition and proceed with explanatory comments on the syntax.

Definition 1 (Syntax). *The sets of values V, of pure values V_0, of programs P, of program types T and of pure program types T_0 are defined by the following grammars (where $n \in \mathbb{N}$, $v \in \mathbf{B}$ (see below), $X \in \Sigma_{\mathbf{B}}$, s denotes strings, and f denotes submarkov kernels from V_0 to V_0):*

$$
\begin{aligned}
V &:= \mathsf{value}\, v \mid \mathsf{var}\, n \mid (V, V) \mid \Lambda P \mid \mathsf{loc}\, n \mid \mathsf{inl}\, V \mid \mathsf{inr}\, V \mid \mathsf{fold}\, V \\
V_0 &:= \mathsf{value}\, v \mid (V_0, V_0) \mid \mathsf{inl}\, V_0 \mid \mathsf{inr}\, V_0 \mid \mathsf{fold}\, V_0 \\
P &:= \mathsf{var}\, n \mid \mathsf{value}\, v \mid \mathsf{fun}(f, P) \mid \Lambda P \mid PP \mid \mathsf{loc}\, n \mid \mathsf{ref}\, P \mid !P \mid P := P \mid \\
&\quad \mathsf{event}\, s \mid \mathsf{eventlist} \mid \mathsf{fold}\, P \mid \mathsf{unfold}\, P \mid (P, P) \mid \mathsf{fst}\, P \mid \mathsf{snd}\, P \mid \\
&\quad \mathsf{inl}\, P \mid \mathsf{inr}\, P \mid \mathsf{case}\ P\ P\ P \\
T &:= \mathrm{Value}_X \mid T \times T \mid T + T \mid T \to T \mid \mathrm{Ref}\, T \mid \mu T \mid \mathsf{Tvar}\, n \\
T_0 &:= \mathrm{Value}_X \mid T_0 \times T_0 \mid T_0 + T_0 \mid \mu T_0 \mid \mathsf{Tvar}\, n
\end{aligned}
$$

In the following we will use the variables P for programs, V for values, T and U for types, where V_0 and T_0, U_0 denote pure values and pure types, respectively.

For a measurable set X, the type of basic values in that set is denoted Value_X. For types T and U, $T \times U$ and $T + U$ denote the pair type and the sum type of T and U, respectively. $T \to U$ denotes the type of functions from T to U, and $\mathsf{Ref}\, T$ denotes the type of references of type T. μT introduces a binder in De Bruijn notation [13] and $\mathsf{Tvar}\, n$ denotes the type variable with De Bruijn index n, i.e., type variables belonging to the $(n+1)$-st enclosing binder μ. For readability we will also use the notation $\mu x.T$ to denote types with named variables. Intuitively the type $\mu x.T$ represents the infinite type which is obtained by recursively substituting x with $\mu x.T$ in T. The set of pure types T_0 consists of the types which do not contain function or reference types.

We do not fix the basic types that are available in our language for extensibility but instead assume a type $\mathbf{B}$ that contains the basic types we expect to need. This includes the type unit with a single element *unit*, and the type of real numbers real, but also functions of type $\mathsf{nat} \to \mathsf{bool}$, which can be used to encode infinite random tapes. The construct $\mathsf{value}\, v$ is used to introduce an element $v \in \mathbf{B}$ in programs.

Programs use De Bruijn notation [13] for variables. With $\mathsf{var}\, n$ we denote variables with De Bruijn index n and with ΛP we denote the function with body P. For readability we will also use the notation $\Lambda x.P$ to denote functions with named variables. Such functions can be converted to unnamed functions in De Bruijn notation using standard techniques. Additionally we write $\mathsf{let}\ x = A\ \mathsf{in}\ B$ for $(\Lambda x.B)A$. Function application is written $P_1 P_2$ where P_1 is the function and P_2 is its argument. Store locations are denoted by $\mathsf{loc}\, n$, reference creation by $\mathsf{ref}\, P$, dereferencing by $!P$, and assignment by $P_1 := P_2$. The language also provides pairs (P_1, P_2) and their projections $\mathsf{fst}\, P$ and $\mathsf{snd}\, P$. Sums are constructed using $\mathsf{inl}\, P$ and $\mathsf{inr}\, P$ and destructed using the $\mathsf{case}\ P_1\ P_2\ P_3$ construct. Events are raised using the construct $\mathsf{event}\, s$. The list of previously raised events is given by $\mathsf{eventlist}$. In an iso-recursive setting (in contrast to an equi-recursive setting) the substitutions of μT into itself (as explained above) are made explicit using the construction $\mathsf{unfold}\, P$ and its inverse $\mathsf{fold}\, P$.

Programs of the form $\mathsf{value}\, v$, $\mathsf{var}\, n$, ΛP, and $\mathsf{loc}\, n$ are considered values. If V_1 and V_2 are values, then so are (V_1, V_2), $\mathsf{inl}\, V_1$, $\mathsf{inr}\, V_1$, and $\mathsf{fold}\, V_1$. Pure values V_0 are values that do not contain variables, Λ-abstractions, and locations.

Using pairs, sums, and recursive types we can define other types like booleans and lists. For example, we set $\mathsf{bool} := \mathrm{Value}_{\mathsf{unit}} + \mathrm{Value}_{\mathsf{unit}}$ for booleans and $\mathrm{list}\, T := \mu x.(\mathrm{Value}_{\mathsf{unit}} + (T \times x))$ for lists over type T. We can then introduce syntactic sugar for conditionals via $\mathsf{if}\ P_1\ \mathsf{then}\ P_2\ \mathsf{else}\ P_3 := \mathsf{case}\ P_1\ (\Lambda \uparrow P_2)\ (\Lambda \uparrow P_3)$. (Here $\uparrow P$ denotes the lifting of all free variables in P by one.) The usual constructors for lists are defined by $nil := \mathsf{fold}\,(\mathsf{inl}\,(\mathsf{value}\, unit))$ and $P_1 :: P_2 := \mathsf{fold}\,(\mathsf{inr}\,(P_1, P_2))$. We further assume the definition of a type char of characters and a type $\mathrm{string} := \mathrm{list}\, \mathrm{char}$.

Probabilism is introduced by the construct $\mathsf{fun}(f, P)$, where f is a submarkov kernel from V_0 to V_0. It is interpreted as applying f to P, yielding a probability measure on pure values. This construct is truly expressive: It allows for expressing every (deterministic) mathematical function g by using the kernel

$f := \lambda x.\delta_{g(x)}$, but also every probabilistic function such as a coin-toss; moreover, it is not even limited to discrete probability measures. Using this construct we can, for example, express the random selection of infinite random tapes. We implemented the language in the proof assistant Isabelle/HOL [27]. The restriction to submarkov kernels on pure values is due to the fact that the datatype for P must not contain functions with argument type P in Isabelle/HOL. The extension Isabelle/HOLCF [26] which includes Scott's logic for computable functions allows such datatypes, but we decided not to introduce this additional domain-theoretic complexity.

The set of program rectangles $\hat{P}$ is defined inductively by including $\{\mathsf{var}\, n\}$ for all $n \in \mathbb{N}$, $\{\mathsf{value}\, v | v \in B\}$ for all $B \in \Sigma_{\mathbf{B}}$, $\{\mathsf{event}\, s\}$ for all s, $\{\mathsf{eventlist}\}$, $\{\mathsf{fun}(f, P) | P \in A \wedge f \in F\}$ for all $A \in \hat{P}$ and arbitrary F, $\{P_1 P_2 | P_1 \in A_1 \wedge P_2 \in A_2\}$ for all $A_1, A_2 \in \hat{P}$, $\{(P_1, P_2) | P_1 \in A_1 \wedge P_2 \in A_2\}$ for all $A_1, A_2 \in \hat{P}$, and analogously for the other cases. We define the σ-algebra of programs Σ_P as the σ-algebra generated by $\hat{P}$.

We model type environments as lists of program types. In the following let Γ be a type environment which is used to assign types to De Bruijn indices (by giving $\mathsf{var}\, n$ the n-th type in Γ) and let Θ be a type environment to assign types to store locations (by giving $\mathsf{loc}\, n$ the n-th type in Θ). We write $\Gamma | \Theta \vdash P : T$ to denote that program P has type T under the variable type environment Γ and the store type environment Θ. The inference rule for this relation are straightforward and thus omitted in this version.

4.3 Semantics of the Language

We now define the semantics of our language. Defining a denotational semantics in the presence of a higher-order store constitutes a highly sophisticated task. Domain-theoretic models have been developed [20,31], but they rely on a complex machinery that turns proving even simple properties in these models into a challenging task.

We thus define an operational small-step semantics $\rightsquigarrow$ of the language. Since the language contains references and is probabilistic, the reduction relation $\rightsquigarrow$ maps program states to distributions over program states. Furthermore, a program state includes a list of previously raised events. We have implemented the semantics using the proof assistant Isabelle/HOL [27], see Section 8.

Definition 2 (Program State and Reduction). *A* program state *is a triple $P|\sigma|\eta$ of a program P, a list σ of programs (the store), and a list η of strings (the events raised so far).*

The reduction relation $\rightsquigarrow$ *is a relation between program states and subprobability measures over program states. It is defined using evaluation contexts as explained by the rules given in Figure 1.*

The rules from Figure 1 are defined using evaluation contexts. An evaluation context E is a program with a hole $\square$, where another program P can be inserted, written $E[P]$. Evaluation contexts have the property that the hole is always at

$$E := \Box \mid \mathsf{fun}(f, E) \mid (E, P) \mid (V, E) \mid EP \mid VE \mid \mathsf{ref}\ E \mid !E \mid (E := P) \mid (V := E)$$
$$\mid \mathsf{fst}\ E \mid \mathsf{snd}\ E \mid \mathsf{fold}\ E \mid \mathsf{unfold}\ E \mid \mathsf{inl}\ E \mid \mathsf{inr}\ E \mid \mathsf{case}\ E\ P\ P \mid \mathsf{case}\ V\ E\ P \mid \mathsf{case}\ V\ V\ E$$

$$E[(\Lambda P)V]|\sigma|\eta \rightsquigarrow \delta_{E[P\{V\}]|\sigma|\eta} \qquad \textsc{Appl}$$
$$E[\mathsf{ref}\ V]|\sigma|\eta \rightsquigarrow \delta_{E[\mathsf{loc}\,(|\sigma|)]|\sigma@[V]|\eta} \qquad \textsc{Ref}$$
$$E[!(\mathsf{loc}\,l)]|\sigma|\eta \rightsquigarrow \delta_{E[\sigma_l]|\sigma|\eta} \quad , \text{if } l < |\sigma| \qquad \textsc{Deref}$$
$$E[\mathsf{loc}\,l := V]|\sigma|\eta \rightsquigarrow \delta_{E[\mathsf{value}\ unit]|\sigma[l:=V]|\eta} \quad , \text{if } l < |\sigma| \qquad \textsc{Assign}$$
$$E[\mathsf{fst}\,(V, V')]|\sigma|\eta \rightsquigarrow \delta_{E[V]|\sigma|\eta} \qquad \textsc{Fst}$$
$$E[\mathsf{snd}\,(V, V')]|\sigma|\eta \rightsquigarrow \delta_{E[V']|\sigma|\eta} \qquad \textsc{Snd}$$
$$E[\mathsf{fun}(f, V)]|\sigma|\eta \rightsquigarrow (\lambda x.\ E[x]|\sigma|\eta)(fV) \qquad \textsc{Fun}$$
$$E[\mathsf{event}\ s]|\sigma|\eta \rightsquigarrow \delta_{E[\mathsf{value}\ unit]|\sigma|\eta@[s]} \qquad \textsc{Ev}$$
$$E[\mathsf{eventlist}]|\sigma|\eta \rightsquigarrow \delta_{E[\overline{\eta}]|\sigma|\eta} \qquad \textsc{EvList}$$
$$E[\mathsf{case}\ (\mathsf{inl}\,V)\ V_L\ V_R]|\sigma|\eta \rightsquigarrow \delta_{E[V_L V]|\sigma|\eta} \qquad \textsc{CaseL}$$
$$E[\mathsf{case}\ (\mathsf{inr}\,V)\ V_L\ V_R]|\sigma|\eta \rightsquigarrow \delta_{E[V_R V]|\sigma|\eta} \qquad \textsc{CaseR}$$
$$E[\mathsf{unfold}\ (\mathsf{fold}\,V)]|\sigma|\eta \rightsquigarrow \delta_{E[V]|\sigma|\eta} \qquad \textsc{Fold}$$

Fig. 1. Reduction rules

the position, where the program should be evaluated first according to a call-by-value strategy. For example to evaluate an application P_1P_2, we want P_1 to be evaluated first, unless it is already a value, in which case we evaluate P_2. This behavior is specified by the context rules EP and VE, i.e., either evaluate the first part, or given that the first part is a value V, evaluate the second part.

Evaluation contexts constitute an elegant way to specify structural rules like where the evaluation should take place first. To specify the non-structural evaluation steps, we use additional rules. To consider the application example again, we have the rule that an abstraction ΛP applied to a value V, evaluates to $P\{V\}$, where $P\{V\}$ denotes the substitution of $\mathsf{var}\,0$ in P with V together with the shifting of all other De Bruijn indices in P by -1. More formally, for all evaluation contexts E, the program state $E[(\Lambda P)V]|\sigma|\eta$ reduces to the Dirac measure $\delta_{E[P\{V\}]|\sigma|\eta}$ (rule Appl).

The rules concerning the store are Ref, Deref, and Assign. The state $E[\mathsf{ref}\ V]|\sigma|\eta$ reduces to $\delta_{E[\mathsf{loc}\,(|\sigma|)]|\sigma@[V]|\eta}$, i.e., the value V is appended to the store and its location $|\sigma|$ is returned. Assuming $l < |\sigma|$, the state $E[!(\mathsf{loc}\,l)]|\sigma|\eta$ reduces to the l-th element of σ, namely $\delta_{E[\sigma_l]|\sigma|\eta}$, and the assignment $E[\mathsf{loc}\,l := V]|\sigma|\eta$ replaces the l-th element of σ by V and returns the value $unit$, namely $\delta_{E[\mathsf{value}\ unit]|\sigma[l:=V]|\eta}$.

The pair destructors $E[\mathsf{fst}\,(V_1, V_2)]|\sigma|\eta$ and $E[\mathsf{snd}\,(V_1, V_2)]|\sigma|\eta$ reduce to the first and the second component of the pair, respectively, namely $\delta_{E[V_1]|\sigma|\eta}$ and $\delta_{E[V_2]|\sigma|\eta}$.

The function construct $E[\mathsf{fun}(f, V)]|\sigma|\eta$ reduces to the distribution obtained by applying f to V and embedding the result into the context and the state, namely we reduce to $(\lambda x.\ E[x]|\sigma|\eta)(fV)$.

If an event is raised, as in $E[\mathsf{event}\ s]||\sigma|\eta$, the event is appended to the event list η and the *unit* value is returned, namely $\delta_{E[\mathsf{value}\ unit]||\sigma|\eta@[s]}$. The program state $E[\mathsf{eventlist}]||\sigma|\eta$ returns the list of previously raised events η, namely it reduces to $\delta_{E[\overline{\eta}]||\sigma|\eta}$. Here $\overline{\eta}$ is the representation of η in the program syntax, i.e., a list of strings.

If the case-construct is applied to a value $\mathsf{inl}\,V$ as in $E[\mathsf{case}\ (\mathsf{inl}\,V)\ V_L\ V_R]||\sigma|\eta$, then the value V is given as an argument to V_L, namely $\delta_{E[V_L\,V]||\sigma|\eta}$. Likewise the program state $E[\mathsf{case}\ (\mathsf{inr}\,V)\ V_L\ V_R]||\sigma|\eta$ reduces to $\delta_{E[V_R\,V]||\sigma|\eta}$.

Finally we need to rule that an unfold applied to a fold cancels out, namely $E[\mathsf{unfold}\ (\mathsf{fold}\,V)]||\sigma|\eta$ reduces to $\delta_{E[V]||\sigma|\eta}$

Similar to the relation $\Gamma|\Theta \vdash P : T$ for programs, we can define a relation $\Gamma|\Theta \vDash P|\sigma|\eta : T$ which types program states. We omit the formal definition in this version. We can now establish the uniqueness, progress and type preservation for the relation $\rightsquigarrow$.

Lemma 3 (Uniqueness, Progress and Preservation). *Let $P|\sigma|\eta$ be a program state, and T a type. Then the following holds:*

- Uniqueness: *There exists at most one measure μ such that $P|\sigma|\eta \rightsquigarrow \mu$.*
- Progress and preservation: *Assume that P is not a value and that $[]||\Theta \vDash P|\sigma|\eta : T$. Then there is a probability measure μ over program states such that $P|\sigma|\eta \rightsquigarrow \mu$ and it holds that*

$$\forall (P'|\sigma'|\eta') \leftarrow \mu. \exists \Theta'.\ ([]|(\Theta @ \Theta') \vDash P'|\sigma'|\eta' : T).$$

We finally define the denotation of a program. First, we define a function $step_n$ that intuitively performs n steps according to $\rightsquigarrow$. Second, the resulting measure is restricted to values, since we only consider execution paths that terminate. Finally, the denotation of a program is given by the supremum of such measures.

Definition 4 (Denotations of Programs). *Let $P|\sigma|\eta$ be a program state, let $step(P|\sigma|\eta) := \mu$ if $P|\sigma|\eta \rightsquigarrow \mu$, and $step(P|\sigma|\eta) := \delta_{P|\sigma|\eta}$ otherwise. Let $step_0(P|\sigma|\eta) := \delta_{P|\sigma|\eta}$ and $step_{n+1} := step \circ step_n(P|\sigma|\eta)$. Let Val be the set of program states $V|\sigma|\eta$ where V is a value.*

Given a program state $P|\sigma|\eta$, define $\mu^n := step_n(P|\sigma|\eta)$ (the distribution of the program state after n steps), let $\bar{\mu}^n := \mu^n|_{Val}$ (the program state after n steps restricted to values only), and define the denotation of $P|\sigma|\eta$ as $[\![P|\sigma|\eta]\!] := \sup_n \bar{\mu}^n$ (the distribution of the program state after the execution). The denotation of a program P is defined as $[\![P]\!] := [\![P|[]|[]]\!]$.

The rules in Figure 1 only employ Dirac measures and submarkov kernels on their right-hand side. From this we can prove that the functions *step* and $\lambda P.[\![P]\!]$ are submarkov kernels.

Lemma 5. *The function step is a submarkov kernel and $\lambda P.[\![P]\!]$ is a submarkov kernel.*

5 Defining Polynomial Runtime

Cryptographic proofs often only assert security if the runtime of the adversary is bounded polynomially in the security parameter. In this section, we give a formal definition of such polynomial-time programs in our language. Since it is unclear what polynomial time means for programs handling non-computational objects such as reals or arbitrary kernels, we exclude such programs from our definition of polynomial-time programs. More precisely, we only allow programs handling *unit*, booleans (or bits), and kernels performing bit-operations. Furthermore, the notion of polynomial-time programs shall not depend on whether events have been raised or not.

Definition 6 ((Non)Computational/Eventless). *A program is a* non-computational atom *iff it is of the form* $\mathsf{eventlist}$, $\mathsf{value}\, v$, *where* $v \neq \mathit{unit}$, *or* $\mathsf{fun}(f, P)$, *where* f *does not compute a coin-toss or one of the bit-operations* $\neg, \wedge, \vee$. *A program is* computational *iff it does not contain any non-computational atoms. A program is* eventless, *iff it does not contain any programs of the form* $\mathsf{eventlist}$ *and* $\mathsf{event}\, s$.

Since the definition of polynomial-time programs should be able to deal with oracles, we have to exclude the time that is spent for executing the oracles, i.e., the notion of polynomial-time shall not depend on the running-time of the oracles the program under consideration calls. We express this by transforming the program so that it raises a distinguished *step event* for every step it takes. If the oracles do not raise events themselves, the running time of the program is defined as the number of step events it raises. Given a program P, the step-annotated program $P!$ is constructed by replacing every sub-term t of P by $(\Lambda{\uparrow}t)(\mathsf{event}\ \mathit{step})$.

If a program takes a pair (V_0, P_1) as argument, we define polynomial-time in the size $|V_0|$ of the first component. Here V_0 is a pure value, e.g., the security parameter. The argument P_1 might contain other arguments that potentially include oracles. A program is defined to be polynomial-time if the number of step events it raises while running $P!$ on (V_0, P_1) is polynomial in $|V_0|$.

Definition 7 (Polynomial-time Programs). *A program* P *of type* $(T_0 \times T) \to U$ *is* polynomial-time, *iff it is computational,* T_0 *is a pure type, and there is a polynomial* q *such that for all inputs* (V_0, P_1) *where* V_0 *is of type* T_0 *and* P_1 *of type* T *is eventless, it holds*

$$\forall n. \forall (P'|\sigma|\eta) \leftarrow \mathit{step}_n(P!\,(V_0, P_1)|[]|[]). \#_{\mathit{step}}(\eta) \leq q(|V_0|),$$

where $\#_{\mathit{step}}(\eta)$ *denotes the number of step events in the event list* η.

For programs of type $T_0 \to U$ that should be polynomial-time in the size of its input, where T_0 is a pure type (i.e., they do not use oracles), we define the notion of first-order polynomial-time: A program P of type $T_0 \to U$ is *first-order polynomial time*, iff the program $\Lambda x.(\uparrow P)(\mathsf{fst}\, x)$ is polynomial-time. We call a program an *efficient algorithm*, iff it is first-order polynomial time, eventless and does not use references.

6 Defining Program Relations

Game-based proofs are conducted by transformation of games (or programs) such that a game and its transformation are in some sense equivalent or indistinguishable. In this section, we show how several such relations can be formalized in our framework. We start with the notion of denotational equivalence, proceed with the notion of observational equivalence, and conclude with the notion of computational indistinguishability.

6.1 Denotational Equivalence

Among the relations we consider, denotational equivalence constitutes the strongest such relation. Two programs P_1 and P_2 are denotationally equivalent if their denotations $[\![P_1]\!]$ and $[\![P_2]\!]$ are the same. This relation is too strong in many cases, since many game transformations do not preserve the denotation completely and introduce small errors. Consider for example two programs P_1 and P_2, where P_1 selects a bitstring uniformly at random from $\{0,1\}^n$ and P_2 from $\{0,1\}^n \setminus \{0^n\}$. It is clear that $[\![P_1]\!] \neq [\![P_2]\!]$, but their statistical difference is very small. The following definition introduces the notion of denotational equivalence up to some error.

Definition 8 (Denotational equivalence up to error). *Two programs* P_1 *and* P_2 *are* denotationally equivalent up to error ε*, iff*

$$\max_S \{|[\![P_1]\!](S) - [\![P_2]\!](S)|\} \leq \varepsilon$$

Denotational equivalence is also too strong from another perspective: Since it compares programs based on the distribution on program states they compute, these states also contain store allocations, which are inaccessible after the execution. Consequently, two programs which compute the same function are denotationally different if, e.g., the first program uses references while the second one does not. Moreover the terms in the distributions have to be equal *syntactically*. Consider the programs $\Lambda x.(\Lambda y.y)x$ and $\Lambda x.x$. Both compute the identity function, but they are denotationally different.

6.2 Observational Equivalence

Comparing programs based on their behavior instead on their denotation yields the notion of *observational equivalence*. The idea is that when used as part of a larger program R, two observationally equivalent programs should be replacable with each other without affecting the computation of R, since it is impossible for R to observe which program it contains. Two programs P and Q are observationally equivalent, if their behavior is equivalent in every context. Here we restrict the set of contexts to those that assign all free variables of P and Q and that do not contain locations outside the store. More formally, we call a program state $P|\sigma|\eta$ *fully closed* if P does not contain free variables, and P and

σ do not contain locations greater that $|\sigma|$. Then two programs are observationally equivalent if for any context that makes them fully closed, the probability of termination is the same.

Definition 9 (Observational Equivalence – Untyped). *Two programs P and Q are* observationally equivalent *if for all contexts C such that $C[P]||[]||[]$ and $C[Q]||[]||[]$ are fully closed, we have that $[\![C[P]||[]||[]]\!](\mathcal{PS}) = [\![C[Q]||[]||[]]\!](\mathcal{PS})$. Here $\mathcal{PS}$ is the set of all program states.*

In many applications, observational equivalence with respect to arbitrary contexts may be too strong a notion. Often it is sufficient to consider contexts that match the type of the programs. More precisely, we say a context has type (Γ, Θ, T) given a hole of type (Γ', Θ', T') if $\Gamma|\Theta \vdash C[P] : T$ whenever $\Gamma'|\Theta' \vdash P : T'$. (This can be concisely formulated by adding the introduction rule $(\Gamma'@\Gamma'')|(\Theta'@\Theta'') \vdash \Box : T'$ to the introduction rules for $\vdash$ and then requiring $\Gamma|\Theta \vdash C : T$ in this modified typing relation.)

Then we can define the typed observational equivalence as follows:

Definition 10 (Observational Equivalence – Typed). *Assume two programs P and Q satisfying $\Gamma|\Theta \vdash P : T$ and $\Gamma|\Theta \vdash Q : T$.*

Then P and Q are observationally equivalent having type (Γ, Θ, T) for contexts of type T'*, if for all contexts such that C has type $([], [], T')$ for holes of type (Γ, Θ, T) it holds that $[\![C[P]||[]||[]]\!](\mathcal{PS}) = [\![C[Q]||[]||[]]\!](\mathcal{PS})$.*

Note that if $\Gamma|\Theta \vdash P : T$ and C has type $([], [], T')$ for holes of type (Γ, Θ, T), we have that $[]||[] \vdash C[P] : T'$ and thus $C[P]||[]||[]$ is fully closed. Hence Definition 10 implies Definition 9.

Observational equivalence can be used to define the notion of efficiently computable functions: A function f is *efficiently computable* if there is an efficient algorithm f', such that f' and the embedding of f as program (using the kernel $\lambda x.\delta_{f(x)}$) are observationally equivalent.

6.3 Computational Indistinguishability

We finally define the notion of computational indistinguishability for families of programs. Intuitively, two families $(P_k)_{k\in\mathbb{N}}$ and $(Q_k)_{k\in\mathbb{N}}$ are computationally indistinguishable if every polynomial-time program D distinguishes P_k and Q_k with at most negligible probability in k.

Definition 11 (Computational Indistinguishability). *A family of programs $(P_k)_{k\in\mathbb{N}}$ is of type T if $[]||[] \vdash P_k : T$ for all $k \in \mathbb{N}$. Let $1^0 := nil$ and $1^{k+1} := \mathsf{value}\, unit::1^k$. Let $\mathcal{S}_{true}$ be the set of program states $V|\sigma|\eta$ where $V = true$ and let $\Pr[P] := [\![P]\!](\mathcal{S}_{true})$ (the probability that P evaluates to true). Two families $(P_k)_{k\in\mathbb{N}}$ and $(Q_k)_{k\in\mathbb{N}}$ of type T are computationally indistinguishable, iff for all polynomial-time programs D the following function is negligible:*

$$\lambda k.|\Pr[D(1^k, P_k)] - \Pr[D(1^k, Q_k)]|$$

7 Fundamental Properties of the Language

7.1 A Chaining Rule for Denotations

The representation of the semantics in Figure 1 suggests that, given a program $E[P]$ where E is an evaluation context, first P is reduced until it is a value V, and then $E[V]$ is reduced until it becomes a value. To reflect that the program P may branch probabilistically and execute a different number of steps in the different branches, we will start to evaluate $E[V]$ at different points in time. In the notation of Definition 4, there is no value n such that μ^n is a distribution over states of the form $E[V]|\sigma'|\eta'$ for some values V and lists σ', η'. However, we can establish the following result in the limit of n: If we compute the denotation $[\![P|\sigma|\eta]\!]$ and apply to this measure the kernel that maps $V|\sigma'|\eta'$ to its denotation, then the result is equal to the denotation $[\![E[P]|\sigma|\eta]\!]$.

Lemma 12 (Chaining denotations). *Let E be an evaluation context (as in Figure 1), P a program, σ a store and η an event list. Then*

$$[\![E[P]|\sigma|\eta]\!] = (\lambda(V|\sigma'|\eta').[\![E[V]|\sigma'|\eta']\!]) \cdot [\![P|\sigma|\eta]\!]$$

This lemma is a powerful tool for reasoning about denotational equivalence. In particular, it directly entails that if P and Q are denotationally equivalent then $E[P]$ and $E[Q]$ are denotationally equivalent as well.

7.2 The CIU Theorem

Establishing observational equivalence of two programs P and Q can be very difficult in general because of the arbitrary context C interacting with P and Q. It would be simpler if we were allowed to quantify only over *evaluation* contexts E as in the following definition (following [22]). For simplicity, we concentrate on the untyped case here.

Definition 13 (CIU Equivalence). *Let P and Q be programs. Given a list a of values, we write P^a for the program term resulting from replacing the i-th free variable in P by a_i. We call P and Q* CIU equivalent *if for all evaluation contexts E, all lists of values a, all stores σ and all event lists η the following holds: If $P^a|\sigma|\eta$ and $Q^a|\sigma|\eta$ are fully closed then $[\![E[P^a]|\sigma|\eta]\!](\mathcal{PS}) = [\![E[Q^a]|\sigma|\eta]\!](\mathcal{PS})$ where $\mathcal{PS}$ is the set of all program states.*

This definition is much easier to handle than Definition 9. For example, assume two denotationally equivalent programs P and Q without free variables and without the subterm eventlist. Then $P^a = P$ and $Q^a = Q$, and it is easy to verify that $[\![P^a|\sigma|\eta]\!] = [\![Q^a|\sigma|\eta]\!]$ for all σ, η. By Lemma 12 we get

$$[\![E[P^a]|\sigma|\eta]\!] = f \cdot [\![P^a|\sigma|\eta]\!] = f \cdot [\![Q^a|\sigma|\eta]\!] = [\![E[Q^a]|\sigma|\eta]\!]$$

where $f(V|\sigma'|\eta') := [\![E[V]|\sigma'|\eta']\!]$. Hence P and Q are CIU equivalent.

Theorem 14 (CIU Theorem). *If P and Q are CIU equivalent, they are observationally equivalent in the sense of Definition 9.*

The proof follows the ideas of [22] with some extensions to cope with the probabilistic nature of our language. ([22] performs an induction over the length of the computation of a terminating program. For probabilistic programs, this length is not necessarily bounded.)

As an immediate corollary we get that denotationally equivalent programs P and Q without free variables or eventlist are also observationally equivalent.[2]

7.3 Exchanging Lines

One of the most elementary transformations on programs is to exchange two lines of code. Yet, in the presence of side effects, changing the order of lines will change the order of execution which in turn will lead to a different program. In this section, we present conditions under which two lines of code can be exchanged in our language. Given programs A, B, C, let $P_{ABC} := \mathsf{let}\ a = A\ \mathsf{in}\ \mathsf{let}\ b = B\ \mathsf{in}\ C(a,b)$ and $P_{BAC} := \mathsf{let}\ b = B\ \mathsf{in}\ \mathsf{let}\ a = A\ \mathsf{in}\ C(a,b)$. Here we assume that C does not directly access a and b, i.e., that it does not have free variables with de-Bruijn indices 0 and 1. Then, if we can derive that P_{ABC} and P_{BAC} are observationally equivalent, we can swap A and B in arbitrary positions in a program. The following theorem gives conditions under which this is possible.

Theorem 15 (Line Swapping). *Assume that A, B, and C do not contain locations, and that A and B do not contain free variables, and that A does not contain* eventlist *or* event*. Then P_{ABC} and P_{BAC} are observationally equivalent.*

The basic idea of the proof is to show that when we execute first A and B on the same initial store, resulting in stores σ_A and σ_B, the store σ_{AB} resulting from applying A and B successively can be computed from σ_A and σ_B as $\sigma_{AB} = f_{AB}(\sigma_A, \sigma_B)$ for a suitable function f_{AB}. Similarly, the store resulting from applying first B and then A is $\sigma_{BA} = f_{BA}(\sigma_A, \sigma_B)$. By showing that the images of f_{AB} and f_{BA} are identical up to a reordering of the store, we have that exchanging A and B only leads to a reordering of the store. From this we get that also $[\![P_{ABC}|\sigma|\eta]\!]$ and $[\![P_{BAC}|\sigma|\eta]\!]$ are identical up to a reordering of the store. Then, using an argument similar to the discussion after Definition 13, we see that P_{ABC} and P_{BAC} are CIU equivalent and thus by Theorem 14 observationally equivalent; this shows Theorem 15.

As locations usually only appear in intermediate execution steps, but not in the programs we reason about, the requirement that A, B, and C do not contain locations does not restrict the applicability of the theorem. However, in some cases one might want A and B to refer to variables assigned in a surrounding context; in this case A and B would have to contain free variables. Yet, in this case we cannot expect P_{ABC} and P_{BAC} to be observationally equivalent in

[2] Although this might seem obvious, it does not hold for all kinds of languages. For example, a language with reflection do not necessarily have this property.

general, as the free variables could be assigned locations that are then shared by A and B. To guarantee that this cannot happen, we need to type the context in order to guarantee that the variables are not assigned locations; this woudl hence require a typed version of Theorem 14.

7.4 The Fundamental Lemma

A common proof technique is the removal of "bad"-event flags, i.e., events that indicate that something unexpected has happened such as a signature was forged. Here two games P_1 and P_2 are syntactically equal up to some points where P_2 raises some event *bad*, written $P_1 \precsim_{\{bad\}} P_2$. The formal definition of this relation is omitted in this version. The fundamental lemma of game-based proofs says that if P_2 raises the event or diverges with probability $\leq \varepsilon$, then P_1 and P_2 are denotationally equivalent up to error ε.

Lemma 16 (Fundamental Lemma). *Let Evs be the set of program states $P|\sigma|\eta$ where η contains the event bad and let T_0 be a pure type. Given programs P_1 and P_2 of type T_0, where $P_1 \precsim_{\{bad\}} P_2$, let $pr_{bad} := [\![P_2]\!](Evs)$ (the probability that P_2 raises the event bad and terminates), $pr_{\perp} := 1 - [\![P_2]\!](\mathcal{PS})$ (the probability that P_2 does not terminate), and $pr_{bad \vee \perp} := pr_{bad} + pr_{\perp}$ (the probability that P_2 raises the event bad or does not terminate). If $pr_{bad \vee \perp} \leq \varepsilon$, then P_1 and P_2 are denotationally equivalent up to error ε.*

8 Embedding the Type System in HOL

In Section 4 we showed the semantics of the language as it is implemented in Isabelle/HOL. The implementation allows for using the inference rules of the typing relation $\vdash$ to prove that some program P has a certain type T. This proof has to be done interactively. Isabelle provides automatic type inference for its own higher-order logic (HOL), i.e., it is not even possible to write an ill-typed HOL term without being rejected by Isabelle's type checker. For example, it is not possible to write the application $f\ f$ in HOL, since f cannot have type $\alpha \to \beta$ and type α simultaneously. On the other hand, writing the application $P\ P$ for some program P in our language is not rejected by Isabelle. The program P has the HOL type $\mathcal{P}$ and the application has the HOL type $\mathcal{P} \to \mathcal{P} \to \mathcal{P}$. Hence Isabelle accepts the program $P\ P$ and infers the HOL type $\mathcal{P}$, even though it is not typeable with respect to $\vdash$. In the following section we depict how our type system can be embedded into HOL. Such an embedding allows us to benefit from Isabelle's automatic type checking, since it prevents us from accidentally writing ill-typed programs. Moreover, the embedding permits more crisp formulations of theorems and definitions, because we do not need to specify all the side conditions that require the used programs to be well-typed. Instead, these conditions come for free. Furthermore, Definitions like Definition 10 that depend on a type are easier to use as the corresponding type can be implicitly inferred.

We start with a brief review of higher-order logic and proceed by showing how our programs can be embedded in this logic.

8.1 Higher-Order Logic

In (simply typed) higher-order logic (HOL) every value in a statement has to be assigned some type. Types may be elementary types (e.g., booleans, integers) or higher-order types. Examples for higher-order types are $\alpha\,\mathsf{set}$, which denotes the type of sets of elements of type α (where α may again be an elementary or higher-order type), $\alpha \to \beta$, which is the type of all functions taking values of type α to values of type β, or $\alpha \times \beta$, which is the type of all pairs in which the first component has type α and the second has type β. For example, in the expression $1 \in \mathbb{N}$, we would have that 1 has type nat (naturals), $\mathbb{N}$ has type $\mathsf{nat\,set}$, and $\in$ is a function of type $\mathsf{nat} \to \mathsf{nat\,set} \to \mathsf{bool}$ (written in infix notation). An example for a statement that does not typecheck is $M \in M$, as $\in$ necessarily has type $\alpha \to \alpha\,\mathsf{set} \to \mathsf{bool}$ for some α, and thus M needs to have types α and $\alpha\,\mathsf{set}$ simultaneously for some α. The major advantages of HOL are its ability to automatically infer types as well as its quite simple logic. Statements written in HOL are usually shorter and easier to read than their counterparts in untyped logics. HOL also allows to state and prove theorems in a polymorphic way: Instead of giving a concrete type to each variable, we can use type variables $\alpha, \beta, \ldots$ in the statement of the theorem; the intended meaning is that this theorem holds for any instantiation of these type variables with concrete types.

An addition to the typesystem of HOL that is implemented in Isabelle/HOL, there exist Haskell-style type classes. A type class introduces constants and constraints that a type has to satisfy. For example, a type class $\mathsf{semigroup}$ might require that for a type α, there is a constant $\circ$ of type $\alpha \to \alpha \to \alpha$ and it holds that $\circ$ fulfills the semigroup axioms. A type α that fulfills the constraints of a type class is called an instance of that type class. Namely, the type nat is an instance of the type class $\mathsf{semigroup}$. The advantage of type classes comes into play when considering polymorphic statements. The type variables in these statements can then be restricted to a given type class; the statement is then expected to hold for all instantiation of the type variables that satisfy the constraints of the type classes. Consider the statement $x \circ (x \circ x) = (x \circ x) \circ x$, where x may have any type α. In general, this statement will be wrong as there might be types in which $\circ$ is not associative. If we restrict α to the type class $\mathsf{semigroup}$, however, the statement becomes true. The main advantage of type classes is that important side conditions (like being a semigroup) can be captured automatically using type inference and do not need to be stated explicitly, leading to shorter and more readable statements.

8.2 Embedding Programs into HOL

After implementing the type of programs as described in Definition 1 in HOL, all programs will be of the same HOL-type. When writing down a program immediately in HOL, we have to explicitly ensure that the program is well-typed with respect to the typing relation $\vdash$ i.e., we lose the advantage of automatic type inference and type checking as supported in HOL. To leverage the power of HOL to our programs, we define a type class $\mathsf{embeddable\_program}$ that introduces the following constants and constraints for a type α:

– Constants prog_type and prog_embedding of types programtype and $\alpha \to$ programterm, respectively.
– The type prog_type is inhabited.
– For any x (of type α) we have $[]|[] \vdash \mathsf{prog\_embedding}(x) : \mathsf{prog\_type}$.

and a type class pure_program that additionally has the following constants and constraints:

– Constant reverse_prog_embedding of type programterm $\to \alpha$.
– For any x (of type α) we have $\mathsf{prog\_embedding}(x) \in V_0$.
– The composition reverse_prog_embedding $\circ$ prog_embedding is the identity.
– prog_embedding and reverse_prog_embedding are measurable.

Using these type classes we can give natural definitions for prog_type, prog_embedding, and reverse_prog_embedding, for elementary types such as nat, bool, etc., as well as for higher-order types such as $\alpha \times \beta$ and α list where α and β are already of type class pure_program. For these definitions, we can then show that nat, bool, etc., as well as $\alpha \times \beta$ and α list are of type class pure_program.

Assuming that α and β are of type class embeddable_program, $\alpha \times \beta$ and α list still fulfill the constraints to be of type class embeddable_program. Moreover we can define prog_type and prog_embedding for function types $\gamma_1 \to \cdots \to \gamma_n$, where the γ_i are of type class pure_program, such that $\gamma_1 \to \cdots \to \gamma_n$ is of type class embeddable_program.

To embed the notion of type environments, which are lists of program types, we introduce the type class env_type that fixes the constant env_types of type T list. We define instances env_nil and (α, β)env_cons, for which env_types is implemented as being [] and $\mathsf{prog\_type}_\alpha \# \mathsf{env\_types}_\beta$, respectively, where # denotes the concatenation of HOL lists, α is of type class embeddable_program and β is of type class env_type.

We now define the main type of the embedding. The type (α, β)program represents the set of all programs that have type $\mathsf{prog\_type}_\beta$ under the variable type environment $\mathsf{env\_types}_\alpha$ and the empty store type environment. It is defined as the set $\{P \in \mathcal{P} | \mathsf{env\_types}_\alpha | [] \vdash P : \mathsf{prog\_type}_\beta\}$. Here α is of type class env_type and β is of type class embeddable_program.

Defining the HOL type of programs now allows for defining programs in HOL. For example, we can define the constant *Var0*, whose representation is the program var 0. This only types in non-empty environments and has the first element in the environment as type. In particular, the HOL type of *Var0* is $((\alpha, \gamma)\mathsf{env\_cons}, \alpha)\mathsf{program}$. Similarly we can define a constant *Lambda* that embeds the ΛP construct. It expects an argument P that types in a non-empty environment, namely being assigned the type $((\alpha, \gamma)\mathsf{env\_cons}, \beta)\mathsf{program}$, and has the return type $(\gamma, \alpha \to \beta)\mathsf{program}$. Following the examples above we are able to embed all language constructs into HOL. Exploiting these newly introduced constants for writing programs, the type-checker of Isabelle will ensure that we cannot write ill-typed programs.

In the present section, we have shown how to embed programs as defined in Section 4 into the type system of HOL. That is, we have defined a single type $\mathcal{P}$ of

programs, then we have defined a type system $\vdash$ on $\mathcal{P}$, and finally we have defined the HOL-type $(\gamma, \alpha)\mathsf{program}$ as the type of all programs of type $\mathsf{prog\_type}_\alpha$. The question arises whether it is necessary to perform this three-step approach. Instead, one might want to directly define a HOL-type $\alpha\,\mathsf{program}$ of programs of type α in terms of smaller types. For example, the type $\alpha\times\beta\,\mathsf{program}$ might be defined as the type containing all terms (P_1, P_2) with P_1 having type $\alpha\,\mathsf{program}$ and P_2 having type $\beta\,\mathsf{program}$. Unfortunately, this approach does not work as, e.g., the type $\alpha\times\beta\,\mathsf{program}$ also has to contain programs P_1P_2 (applications) with P_1 of type $\gamma \to \alpha\times\beta\,\mathsf{program}$ and P_2 of type $\gamma\,\mathsf{program}$ (for any type γ). Hence the definition of the HOL-type $\alpha\times\beta\,\mathsf{program}$ has to depend (*i*) on an infinite number of other types and (*ii*) on larger types. Such constructions are not supported by HOL and would need much more elaborate (and thus more complicated) type systems than HOL.

8.3 Syntax

The HOL-embedding explained above offers automatic type checking, but it is still not very convenient to actually program in this language. For example, one does not want to write *Lambda Var0* to program the identity function. We overcome this problem by implementing parse and print translations in Isabelle; this allows us to program in an ML-style syntax. We are able to hide the De Bruijn implementation of variables and use named abstractions instead. For example the identity function above is written as $\text{¨}\lambda x.x\text{¨}$. We are also able to express syntactic sugar for constructs that are not present in the underlying language. In particular, we offer a sequence construct $\text{¨}P_1; P_2\text{¨}$, a let-like construct $\text{¨}\mathsf{select}\ x \leftarrow P_1\ \mathsf{in}\ P_2\text{¨}$, as well as pattern matching to write, e.g., $\text{¨}\lambda(x, y).x\text{¨}$. (The quotes $\text{¨}P\text{¨}$ inform Isabelle that P should be parsed as a program and not as a mathematical expression.) We also need the antiquotations $:P:$ and $\hat{}v$. The syntax $:P:$ tells Isabelle that P is to be parsed using the normal mathematical syntax and is supposed to evaluate to a program. The syntax $\hat{}v$ denotes that v is to be parsed using normal mathematical syntax, and the resulting value is to be embedded into the language (e.g., $\hat{}\mathit{true}$ would take the Isabelle-constant *true* and convert it to the corresponding constant in our programming language. Finally, to capture a common pattern in cryptographic definitions, we define $\Pr[B : P]$ as $\text{¨}\mathsf{select}\ P\ \mathsf{in}\ B\text{¨}(\{\mathit{True}\})$. (The probability of reducing to *true*.)

9 Examples

This section gives an illustrating example application of our language. We define encryption schemes and the security notion of indistinguishability under adaptive chosen ciphertext attacks (IND-CCA2).

An encryption scheme consists of three programs. The first program, the key generation Gen, expects a bitstring, namely a list of booleans, as argument and returns a pair $(\mathsf{pk}, \mathsf{sk})$ consisting of a public key pk of type α and a secret key sk of type β. It has type $(\mathsf{env\_nil}, \mathsf{bitstring} \to \alpha \times \beta)\mathsf{program}$. The

second program, the encryption algorithm Enc, expects a public key and a message of type μ as argument and returns a cipher text of type γ. It has type $(\mathsf{env\_nil}, (\alpha \times \mu) \to \gamma)\mathsf{program}$. The third program, the decryption algorithm Dec, computes a message from a secret key and a ciphertext. It has type $(\mathsf{env\_nil}, (\beta \times \gamma) \to \mu)\mathsf{program}$.

Having specified encryption schemes, we can define what it means for an encryption scheme to be correct. This is the case if Gen, Enc, and Dec are first order polynomial time and for all $n \in \mathbb{N}$ and for all messages m it holds that if $(\mathsf{pk}, \mathsf{sk})$ is output by $\mathsf{Gen}(1^n)$, c is the encryption of m using pk and m' is the decryption of c using sk, then the probability that $m = m'$ is 1. In Isabelle, the definition takes the following form (using the embedding and the syntax introduced in Section 8):

$$
\begin{aligned}
&(\mathit{first\_order\_polynomial\_time}\ \mathsf{Gen}) \wedge \\
&(\mathit{first\_order\_polynomial\_time}\ \mathsf{Enc}) \wedge \\
&(\mathit{first\_order\_polynomial\_time}\ \mathsf{Dec}) \wedge \\
&(\forall nm.\ \mathrm{Pr}[(\hat{}\mathsf{op} =)\hat{}m\ m' : (pk, sk) \leftarrow :\mathsf{Gen}{:}(:\mathit{unary\_parameter}\ n{:}); \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad c \leftarrow :\mathsf{Enc}{:}(pk, \hat{}m); \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad m' \leftarrow :\mathsf{Dec}{:}(sk, c)] = 1)
\end{aligned}
$$

The security notion of indistinguishability under adaptive chosen ciphertext attacks (IND-CCA2) says that no adversary can distinguish between the encryptions of two self-chosen messages better than with negligible probability, even if it has access to a decryption oracle. The only restriction is that it is not allowed to query for the decryption of the challenge ciphertext. We model such an adversary as two programs A_1 and A_2, where A_1 outputs the challenge message pair (m_1, m_2) and a string a, which is used for communication between A_1 and A_2. We call A_1 a *message pair adversary*, if all its possible outputs consist of a triple (m_1, m_2, a), where $|m_1| = |m_2|$. We define two decryption oracles. The first oracle D decrypts every ciphertext it is queried on. It is defined as $\ddot{}\lambda\mathsf{sk}.\lambda c.{:}\mathsf{Dec}{:}(\mathsf{sk}, c)\ddot{}$. The second oracle D' behaves similarly, except that it refuses to decrypt the challenge ciphertext c', in which case it returns a default value. It is defined as $\ddot{}\lambda(\mathsf{sk}, c').\lambda c.\mathsf{if}\ (\hat{}\mathsf{op} =)\hat{}c\ c'\ \mathsf{then}\ \hat{}\mathit{default\_value}\ \mathsf{else}\ {:}\mathsf{Dec}\ \mathsf{ES}{:}(\mathsf{sk}, c)\ddot{}$. Given an adversary consisting of two programs A_1 and A_2, respectively, we can define the IND-CCA2 game for a bit b, adversaries A_1 and A_2, and security parameter n as follows:

Definition 17 (IND-CCA game)

$$
\begin{aligned}
&\mathit{IND\_CCA}\ A_1\ A_2\ b\ n' = \\
&\qquad \mathrm{Pr}\big[{:}A_2{:}\ (n, pk, m_1, m_2, c, a, {:}D'{:}(sk, c))\ : \\
&\qquad\qquad\qquad \mathsf{select}\ n \leftarrow :\mathit{unary\_parameter}\ n'{:}; \\
&\qquad\qquad\qquad (pk, sk) \leftarrow :\mathsf{Gen}{:}\ n; \\
&\qquad\qquad\qquad (m_1, m_2, a) \leftarrow :A_1{:}\ (n, pk, {:}D{:}\ sk); \\
&\qquad\qquad\qquad c \leftarrow :\mathsf{Enc}{:}(pk, \mathsf{if}\ \hat{}b\ \mathsf{then}\ m_1\ \mathsf{else}\ m_2)\big]
\end{aligned}
$$

Here unary_parameter is a function (not a program) mapping an integer n' to a bitstring consisting of n ones.

We then define IND-CCA2 security for an encryption scheme ES as follows.

Definition 18 (IND-CCA2-Security). *For all polynomial-time programs A_1 and A_2, where A_1 is a message pair adversary, the following function is negligible.*

$$\lambda n.|(\mathit{IND\_CCA}\ A_1\ A_2\ \mathit{true}\ n) - (\mathit{IND\_CCA}\ A_1\ A_2\ \mathit{false}\ n)|$$

This definition can also be extended to the random oracle model, in which case the random oracle is given to all programs as an additional argument.

We encourage the reader to compare the definition using our language with a typical semi-formal definition from the standard textbook [14] (to make the comparison simpler, we have adapted the definition by removing the auxiliary input and adapting the variable names to match ours):

Definition 19 (IND-CCA security, following [14]). *A public-key encryption scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is said to be IND-CCA secure if for every pair of probabilistic polynomial-time oracle machines, A_1 and A_2, we have that $|\mathit{IND\_CCA}_n^{(1)} - \mathit{IND\_CCA}_n^{(2)}|$ is negligble where*

$$\mathit{IND\_CCA}_n^{(i)} := \Pr\left[\begin{array}{ll} v = 1 & \textit{where} \\ & (pk, sk) \leftarrow \mathsf{Gen}(1^n) \\ & (m^{(1)}, m^{(2)}, a) \leftarrow A_1^{D_{sk}}(1^n, pk) \\ & c \leftarrow \mathsf{Enc}_{pk}(m^{(i)}) \\ & v \leftarrow A_2^{D_{sk}}(a, c) \end{array}\right]$$

where $|m^{(1)}| = |m^{(2)}|$. When given the challenge c, machine A_2 is not allowed to make the query c to the decryption oracle D_{sk}.

10 Conclusion and Future Work

We implemented in Isabelle/HOL a probabilistic higher-order functional language with recursive types, references, and events which is able to express the constructs that typically occur in cryptographic specifications. Is it simple enough to be understandable even for researchers without a strong background in the theory of programming languages.

We are currently working on defining a collection of game transformations that preserve the relations defined in Section 6. These game transformations will culminate in a tool that enables cryptographers to conduct their proofs without having to bother with the details and intricacies of our language, hence making a substantial step towards the vision outlined in [15]. We have started to provide a graphical user interface where simple game transformations are performed using mouse gestures like drag-and-drop.

Acknowledgments. We thank Gilles Barthe and Cătălin Hriţcu for valuable discussions. This work was partially supported by the Cluster of Excellence "Multimodal Computing and Interaction".

References

1. Abadi, M., Gordon, A.D.: A calculus for cryptographic protocols: The spi calculus. Information and Computation 148(1), 1–70 (1999)
2. Abadi, M., Rogaway, P.: Reconciling two views of cryptography: The computational soundness of formal encryption. In: Watanabe, O., Hagiya, M., Ito, T., van Leeuwen, J., Mosses, P.D. (eds.) TCS 2000. LNCS, vol. 1872, pp. 3–22. Springer, Heidelberg (2000)
3. Backes, M., Pfitzmann, B.: Symmetric encryption in a simulatable Dolev-Yao style cryptographic library. In: Proc. 17th IEEE Computer Security Foundations Workshop (CSFW), pp. 204–218 (2004)
4. Backes, M., Pfitzmann, B., Waidner, M.: A composable cryptographic library with nested operations (extended abstract). In: Proc. 10th ACM Conference on Computer and Communications Security, pp. 220–230 (2003); Full version in IACR Cryptology ePrint Archive 2003/015 (January 2003), http://eprint.iacr.org/
5. Barthe, G., Gregoire, B., Janvier, R., Zanella Beguelin, S.: Formal certification of code-based cryptographic proofs. IACR ePrint Archive (August. 2007), http://eprint.iacr.org/2007/314
6. Basin, D., Mödersheim, S., Viganò, L.: OFMC: A symbolic model checker for security protocols. International Journal of Information Security (2004)
7. Bellare, M., Rogaway, P.: The security of triple encryption and a framework for code-based game-playing proofs. In: Vaudenay, S. (ed.) EUROCRYPT 2006. LNCS, vol. 4004, pp. 409–426. Springer, Heidelberg (2006), http://eprint.iacr.org/2004/331.ps
8. Blanchet, B.: A computationally sound mechanized prover for security protocols. In: Proc. 27th IEEE Symposium on Security & Privacy, pp. 140–154 (2006)
9. Blanchet, B., Pointcheval, D.: Automated security proofs with sequences of games. In: Dwork, C. (ed.) CRYPTO 2006. LNCS, vol. 4117, pp. 537–554. Springer, Heidelberg (2006)
10. Canetti, R., Herzog, J.: Universally composable symbolic analysis of mutual authentication and key exchange protocols. In: Halevi, S., Rabin, T. (eds.) TCC 2006. LNCS, vol. 3876, pp. 380–403. Springer, Heidelberg (2006)
11. Corin, R., den Hartog, J.: A probabilistic hoare-style logic for game-based cryptographic proofs. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) ICALP 2006. LNCS, vol. 4052, pp. 252–263. Springer, Heidelberg (2006)
12. Cortier, V., Warinschi, B.: Computationally sound, automated proofs for security protocols. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 157–171. Springer, Heidelberg (2005)
13. de Bruijn, N.G.: Lambda Calculus notation with nameless dummies: a tool for automatic formula manipulation. Indagationes Mathematicæ 34, 381–392 (1972)
14. Goldreich, O.: Foundations of Cryptography, May 2004. Basic Applications, vol. 2. Cambridge University Press, Cambridge (May 2004), http://www.wisdom.weizmann.ac.il/~oded/frag.html
15. Halevi, S.: A plausible approach to computer-aided cryptographic proofs. Cryptology ePrint Archive, Report 2005/181 (2005), http://eprint.iacr.org/
16. Halmos, P.R.: *Measure Theory*. Graduate Texts in Mathematics, vol. 18. Springer, Heidelberg (1974)
17. Kemmerer, R.: Analyzing encryption protocols using formal verification techniques. IEEE Journal on Selected Areas in Communications 7(4), 448–457 (1989)

18. Laud, P.: Semantics and program analysis of computationally secure information flow. In: Sands, D. (ed.) ESOP 2001. LNCS, vol. 2028, pp. 77–91. Springer, Heidelberg (2001)
19. Laud, P.: Symmetric encryption in automatic analyses for confidentiality against active adversaries. In: Proc. 25th IEEE Symposium on Security & Privacy, pp. 71–85 (2004)
20. Levy, P.B.: Possible world semantics for general storage in call-by-value. In: Bradfield, J.C. (ed.) CSL 2002 and EACSL 2002. LNCS, vol. 2471, pp. 232–246. Springer, Heidelberg (2002)
21. Lowe, G.: Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In: TACAS 1996. LNCS, vol. 1055, pp. 147–166. Springer, Heidelberg (1996)
22. Mason, I., Talcott, C.: Equivalence in Functional Languages with Effects. Journal of Functional Programming 1(3), 287–327 (1991)
23. Meadows, C.: Using narrowing in the analysis of key management protocols. In: Proc. 10th IEEE Symposium on Security & Privacy, pp. 138–147 (1989)
24. Micciancio, D., Warinschi, B.: Soundness of formal encryption in the presence of active adversaries. In: Naor, M. (ed.) TCC 2004. LNCS, vol. 2951, pp. 133–151. Springer, Heidelberg (2004)
25. Millen, J.K.: The interrogator: A tool for cryptographic protocol security. In: Proc. 5th IEEE Symposium on Security & Privacy, pp. 134–141 (1984)
26. Müller, O., Nipkow, T., von Oheimb, D., Slotosch, O.: HOLCF = HOL + LCF. Journal of Functional Programming 9(2), 191–223 (1999)
27. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
28. Nowak, D.: A framework for game-based security proofs. IACR Cryptology ePrint Archive 2007/199 (2007), http://eprint.iacr.org/
29. Paulson, L.: The inductive approach to verifying cryptographic protocols. Journal of Cryptology 6(1), 85–128 (1998)
30. Pierce, B.C.: Types and programming languages. MIT Press, Cambridge (2002)
31. Schwinghammer, J.: Reasoning about Denotations of Recursive Objects. PhD thesis, Department of Informatics, University of Sussex, Brighton, UK (July 2006)
32. Shoup, V.: Sequences of games: A tool for taming complexity in security proofs. IACR ePrint Archive (November 2004), http://eprint.iacr.org/2004/332.ps
33. Thayer Fabrega, F.J., Herzog, J.C., Guttman, J.D.: Strand spaces: Why is a security protocol correct? In: Proc. 19th IEEE Symposium on Security & Privacy, pp. 160–171 (1998)
34. The Coq development team. The Coq Proof Assistant Reference Manual (2006), http://coq.inria.fr

Reasoning Using Knots*

Thomas Eiter, Magdalena Ortiz, and Mantas Šimkus

Institute of Information Systems, Vienna University of Technology
{eiter,ortiz,simkus}@kr.tuwien.ac.at

Abstract. The deployment of Description Logics (DLs) and Answer Set Programming (ASP), which are well-known knowledge representation and reasoning formalisms, to a growing range of applications has created the need for novel reasoning algorithms and methods. Recently, knots have been introduced as a tool to facilitate reasoning tasks in extensions of ASP with functions symbols. They were then also fruitfully applied for query answering in Description Logics, hinging on the forest-shaped model property of knowledge bases. This paper briefly reviews the knot idea at a generic level and recalls some of the results obtained with them. It also discusses features of knots and relations to other reasoning techniques, and presents issues for further research.

1 Introduction

In the last years, there has been increasing interest in knowledge representation and reasoning formalisms based on computational logic. Two prominent examples of them are Description Logics (DLs) [2], which play a fundamental role in formal ontology management, and Answer Set Programming (ASP) [4], recognized as a powerful declarative problem solving approach. These formalisms are deployed to a growing range of applications, which in turn raise new requirements and a need for reasoning services that are beyond traditional ones.

In Description Logics, the traditional reasoning tasks include testing satisfiability of a knowledge base, concept subsumption and instance checking; the last two are in fact reducible to satisfiability testing. Recent applications of DLs in ontology and data management, however, require reasoning services that lack this reducibility property and call for dedicated algorithms. In particular, answering queries to a knowledge base such as *conjunctive queries* –which are fundamental in databases– has emerged as a relevant task (see e.g. [9,16,17,19,26]).

In ASP, which is rooted in non-monotonic logic programming, traditional reasoning tasks include construction of some or all answer sets (i.e., models) of a knowledge base, as well as brave/cautious entailment from the models (i.e., truth in some or all models, respectively). In order to ensure finite models, standard ASP solvers disallow function symbols or restrict their use in a suitable manner.

* This work has been partially supported by the Austrian Science Fund (FWF) grants P20840 and P20841, and the Mexican National Council for Science and Technology (CONACYT) grant 187697.

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 377–390, 2008.

As this hinders modeling infinite domains and objects (like evolving processes), respective extensions of ASP or more liberal use of function symbols are desired (see [5,6,7,33] for very recent works and discussions).

Furthermore, beyond novel reasoning services, also the need for extensions of the formalisms has emerged. In particular, the combination of rules and ontologies in *hybrid knowledge bases*, which integrate Logic Programming and Description Logics, is an ongoing research issue, which is non-trivial given that their basic settings are quite different (see e.g. [10,31,24] for more information).

With this background, *knots* were presented in [33] as a tool to solve reasoning tasks in ASP extended with function symbols, that hinges on the properties of a syntactically restricted program class: models are forest-shaped, i.e., collections of trees (not in a strict sense) that are connected in a limited way; noticeably, the models are infinite, but bear certain regularity. Loosely speaking, knots are patterns of model subtrees of depth at most one, which can be instantiated and combined to construct some model(s) of a given program. In fact, the whole set of a models of the program can be represented by a suitable knot set; this was exploited by several reasoning algorithms in [33].

Triggered by the fact that several Description Logics have models of similar structure, knots were then (suitably adapted) transferred to the context of DLs in [28,27], where they could be fruitfully applied to obtain (worst-case) optimal algorithms for conjunctive query answering, providing an exponential improvement w.r.t. previous approaches. This, together with some attractive features, demonstrates the usefulness of knots, and suggests to explore them for further applications, possible within other contexts.

In the next section, we present the knot idea a bit more in detail. We then mention some related reasoning techniques, including tableaux rules, dominoes, mosaics and automata, and highlight some specific features of knots in comparison to them. Special consideration will be given to mosaics, which are perhaps most closely related to knots. In Section 3, we then briefly consider knots in Logic Programming and in DLs, recalling some recent results. Section 4 presents some further issues and concludes this paper.

2 Model Representation Using Knots

2.1 The Knot Technique

The purpose of *knots*, which were introduced in [33] for reasoning over logic programs with function symbols (see Section 3.1 for more details), is the finite representation of possibly infinite tree-shaped structures by means of finite sets of building blocks. Informally, we assume here a given *knowledge base (KB)* in a (first order) language of unary and binary predicates (it can be, for example, a logic program or a recast DL ontology). The KB represents a set of structures, its *models*, which can be naturally represented as labeled graphs. We also assume that these graphs are actually trees or sets of trees, and that their nodes are *terms* with unary function symbols only (intuitively, $f(t)$ is seen as a child node

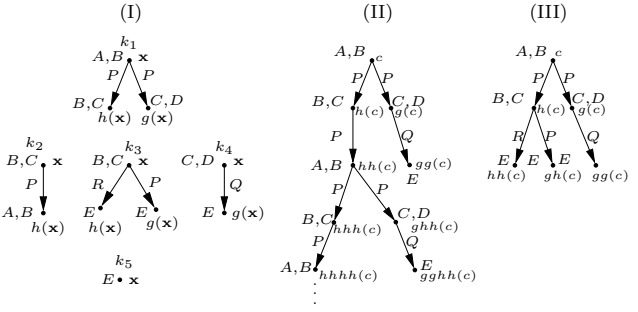


Fig. 1. Example knot set (I) and two generated trees (II) and (III)

of the term t). We will refer to *tree-shaped models* in this loose setting. For formal definitions of (variants of) these assumptions, see [33,28,27].

Essentially, a knot is a small schematic labeled tree of depth ≤ 1, which is a basic 'block' for building tree-shaped models. It is a *pattern* for small subtrees that occur in actual models: such a subtree and the respective knot agree on all the labels, but agree only partially on the structure of their terms. The term at the root and inside the outermost function symbol of its children may differ. Abstracting this difference, we obtain a subtree pattern; in turn, we can instantiate this pattern with different terms and reobtain the subtrees.

Figure 1 illustrates this superficially on an example. On the left hand side, (I) depicts a set of knots $k_1, \ldots, k_5$. We use a constant $\mathbf{x}$, not in the signature of the KB, as their root term. It acts as a 'place-holder' that can be instantiated by any functional term. These knots represent a set of tree-shaped structures that can be built out of them, like the ones depicted in (II) and (III). The tree in (II) starts with the knot k_1, but instead of the 'abstract' $\mathbf{x}$, it contains a constant c from the KB. It is followed by the knots k_2 and k_4, instantiated with $h(c)$ and $g(c)$ respectively. Then the left branch has another instance of k_1, and the sequence is repeated (possibly into the infinite). The finite tree-shaped model on the right contains only one instance of each of the knots k_1, k_3 and k_4.

The general idea underlying the knot technique is that sets of relevant knots can be extracted from the model(s) of the KB and used as their compact representation. Since only finitely many knots exist (over the signature of the KB), knot sets are finite. Some models (or ideally, all models) can be built from these sets by instantiating and combining knots in a suitable manner.

However, in order for a knot set to correctly represent models, some conditions must be fulfilled. They can be grouped into two types:

1. *local consistency conditions*, which apply to each knot individually and ensure that it represents an abstract domain element with its immediate successors in a tree model, as required by the constraints given in the KB.
2. *global conditions*, which apply to the (whole) knot set and ensure that suitable instances of the knots in the set can be composed into full models.

It is desired that both types of conditions can be checked easily, and that global conditions, as an important feature, support an incremental model-building procedure that is *backtracking-free*, i.e., legal choices for knot instances at some point can not lead to inconsistency later.

In case a knot set satisfies both the local and the global conditions, it can be viewed as *coherent*. Coherence of a set of knots is a sufficient condition for it to represent some model(s) of the KB. Furthermore, if this property is preserved under unions of coherent sets, and since there are finitely many distinct knots, there exists a unique set-inclusion maximal coherent set of knots, i.e., the smallest coherent knot set that includes all coherent knot sets. Such maximal set can be seen as *complete*, since it captures all the model structures that can be generated by coherent knot sets (e.g., answer sets of ASP programs, canonical models for query answering in some DLs). In particular applications it might be handy to consider more refined notions of completeness, e.g., completeness with respect to structures satisfying a given condition (e.g., w.r.t. finite structures).

Satisfiability testing of a KB can be reduced to finding a coherent knot set, or, alternatively, to checking whether the maximal coherent knot set is non-empty; this can be done using a simple elimination method, provided certain conditions apply (see Section 4). Further, since coherent knot sets represent multiple (or even all) models, they are not only useful for deciding KB satisfiability. As we will see, they can be fruitfully used to solve various reasoning tasks, including non-standard ones that are hard to solve with more traditional approaches.

2.2 Other Techniques

Many of the basic intuitions that underlie the knot technique can be found in other reasoning methods, specially in the context of (fragments of) first-order logic (FOL). For example, in modal and description logics, which usually can be viewed as fragments of FOL, tree automata have been widely used (see, e.g., [3,4,8,9]). Apart from exploiting the tree-model property of many of these logics, like knots do, they also consider *local* conditions in the form of states and transitions, and *global* acceptance conditions on the runs of the automata. They exploit the finite number of finite 'pieces' that have to be considered to effectively decide modelhood of potentially infinite structures. These relations are even more clear in the case of *mosaics* and *type elimination* methods, which can be viewed as a generalization of tree automata and are very closely related to knots (see Section 2.4 for more details.)

Other methods that are based on different principles still exhibit some common features with knots. For example, we can consider resolution and tableau algorithms, which are both well-known in FOL theorem proving and have been widely explored for modal logics and description logics, (see, e.g., [3,19,26]). In both cases we find some local conditions that define, for a specific pair of clauses or for a given element of a tableau, which inference rules are applicable. There are also some global consistency (clash-freeness, absence of the empty clause) conditions and termination (completeness, saturation) conditions; the latter exploit the fact that only a finite numbers of objects (distinct nodes, clauses) are

relevant to decide the consistency of a KB. Clearly, tableaux and knots are also related in the sense that they closely resemble the models of the KB. On the other hand, a tableau naturally represents one model, while knots represent sets of models. As for resolution, it can be considered a *proof-oriented* technique, in contrast to the *model-oriented* knots. We could say that, unlike knots, resolution is based on the syntax and does aim at representing the structure of models.

In the setting of logic programs for non-monotonic reasoning, specially with function symbols, knots are rather novel and not easily compared with existing reasoning techniques – in part because there are fewer of them as there are for FOL. This, in turn, has several reasons. First, ASP with function symbols is highly undecidable [22,20,21] and, until recently, not many decidable fragments had been considered. Research in ASP focused on the function-free (Datalog) setting, in which most reasoning algorithms have been based on efficient *grounding*. A second aspect is the need for 'model building', which is complicated in general. Unlike FOL-style reasoning problems that can be solved by simply deciding the existence of a (counter)-model, a standard reasoning task in ASP is to construct a set of intended models. A further issue is that, to ensure the stability of a model, some minimality condition must be tested, and this makes reasoning harder in general (e.g., consistency testing for disjunctive propositional programs under the answer set semantics is Σ_2^P-complete as compared with NP-completeness in case of classical semantics). Indeed, it involves testing a global condition that quantifies over all submodels of each candidate interpretation, a property that goes beyond standard first order logic. This is specially challenging in the presence of function symbols, since the candidate models to be considered are infinite in general. Note that deciding whether an arbitrary logic program with functions symbols has some stable model is Σ_1^1-complete.

Most of the existing reasoners build on efficient grounders and use tailored model building and minimality checking algorithms, whose adaptability to the infinite setting is unclear. Alternative approaches have considered, for example, reductions of ASP to satisfiability of propositional and quantified Boolean formulas. However, these have also been limited to the function free setting, and the reasoning methods for ASP with function symbols remain largely undeveloped.

2.3 Features

We briefly summarize some important features and properties of knots which suggest them as a useful technique.

- Knots already proved to be very helpful for deriving novel decidability and tight complexity bounds for formalisms enjoying forest-shaped models. This is because knots bring us close to the structure of models, and allow us to manipulate them more easily.
- The representation of models via knots provides a basis for developing reasoning algorithms that work directly on knot sets instead of the input KBs. Such algorithms can access the sets of knots for various tasks like constructing models or evaluating different kinds of queries.

• These algorithms usually exhibit certain *modularity*. For example, maximal coherent knot sets can be computed independently of the extensional data. This proves specially useful when considering *data complexity* aspects.
• Further, sets of knots are useful for *knowledge compilation*. Indeed, once a suitable knot set is computed (which can be done off-line, in a preprocessing phase), it can be used in the on-line phase to provide more efficient reasoning.
• The compilation can provide the benefit of *adaptability*: as small changes in the KB can often be reflected as small changes in the computed knot sets, algorithms building on them are often re-usable, even in relatively changing environments.
• The model-building infrastructure provided by the knot technique is especially important for the ASP context. For example, in tasks like diagnosis, planning and configuration, possible models are built to show possible failures, plans, etc.
• The capability of knot sets to finitely represent sets of relevant models is useful in the cases when a given query cannot be expressed in the underlying formalism, e.g., the case of conjunctive queries in Description Logics (see Section 3.2).

On the downside of the approach is the exponential space that is required in the worst-case to represent knot sets. However, for the ASP context this space requirement is somewhat inherent, as model construction and traversal are important tasks. The high space requirements are also inherent to the idea of knowledge compilation, and in this case, its cost is amortized over time through the more efficient solution of several instances. Finally, we note that the size of a knot set representing a set of models depends directly on the structural complexity of the represented models; when only few knots occur in them (i.e., they are highly regular), the respective knot set is rather small (note that even small sets may actually encode infinitely many infinite models).

2.4 Knots and Mosaic Techniques

The knot technique is a close relative of mosaics, which were introduced in [25] to prove decidability of equational theories of classes of algebras of relations. The main idea behind them was to show that model existence can be equivalently checked by deciding the existence of a finite set of model-pieces that fulfill certain coherence conditions which allow to reconstruct the model. The knot technique can in fact be seen as a particular mosaic technique that is tailored for ASP and uses knots to build tree-shaped interpretations. The method for deriving maximal coherent knot sets described in [33,28,27] is closely related to type elimination, first described in [29], which was used in an exponential time (worst case optimal) algorithm for Propositional Dynamic Logic (PDL).

More recently, mosaics have been applied to obtain decidability and tight complexity results for some fragments of FOL that are highly relevant for Knowledge Representation. Among them are the guarded fragment of FOL [1], the 2-variable fragment $\mathcal{C}^2$ of FOL [18], and $\mathcal{C}^2$ augmented with counting quantifiers [30]. These fragments capture several modal and description logics; e.g., the last one captures the DL $\mathcal{SHOIQ}$ that underlies the OWL Web Ontology standard.[1] As

[1] This holds after the standard elimination of transitive roles.

a matter of fact, rather than by this relation, our deployment of knots to DLs was motivated by forest-shaped model properties which they share with related classes of logic programs, for which knots were originally introduced.

The knot technique provides a bridge between tree automata and mosaics. Indeed, a single knot can be seen as a transition in an automaton: its nodes are states, and given the root state, the automaton moves into the states given by the leaves. In this setting, the emptiness test of the maximal coherent knot set mimics the emptiness test of the automaton. The mosaic technique is to some extent more liberal than knots and tree automata, since mosaics can deal with models that are not necessarily tree-shaped.

3 Applications

3.1 Knots in Logic Programming

The knot technique has been fruitfully applied to provide decidability and complexity results for the class of $\mathbb{FDNC}$ logic programs with function symbols [33]. Such programs feature *negation as failure* under the Answer Set semantics [15] and follow the line of efforts that augment ASP with function symbols in order to better support common-sense reasoning over infinite processes and objects. As function symbols lead to high undecidability in general, some authors have tried to identify expressive fragments of lower complexity (see, e.g., [5,7]).

The syntax and the Answer Set semantics of $\mathbb{FDNC}$ programs make them especially suitable for reasoning about evolving action domains. Function symbols allow to generate infinite time-lines (with possible branching), while the availability of non-monotonic negation allows to expresses inertia information, i.e., the truth value of a fluent (changeable property) is carried over to follow-up stages in case there is no indication of the opposite. These features allow $\mathbb{FDNC}$, for example, to capture the core of propositional $\mathcal{K}$, a planning language based on the Answer Set semantics for logic programs [11].

As an example, the $\mathbb{FDNC}$ program in Figure 2 represents a well-known shooting scenario illustrating the *frame problem* connected with *inertia* (which properties remain unchanged if a certain action is taken). The rule (1) is an initialization fact saying that in the initial situation the gun is not loaded. The rules (2)-(3) describe the loading action, which occurs if the gun is unloaded and there is no evidence (from a rule) for the target being hit; its effect is a loaded gun. The rule (4) states the inertia of the fluent *Loaded*; it remains true during the transitions unless there is evidence that the fluent *Unloaded* is true. The rules (5)-(6) describe the targeting action that occurs after loading the gun. The rules (7)-(9) describe the shooting action; the agent shoots when the gun is loaded and the object is on target. Its effects are non-deterministic: the target is either hit or missed, and the gun is emptied.

The program has infinitely many answer sets corresponding to possible evolutions of the initial situation. Indeed, if the shooter misses the target, the load-target-shoot process is repeated, and can be repeated forever if she never hits the target. A possible evolution depicted in (I) of Figure 2 corresponds to hitting

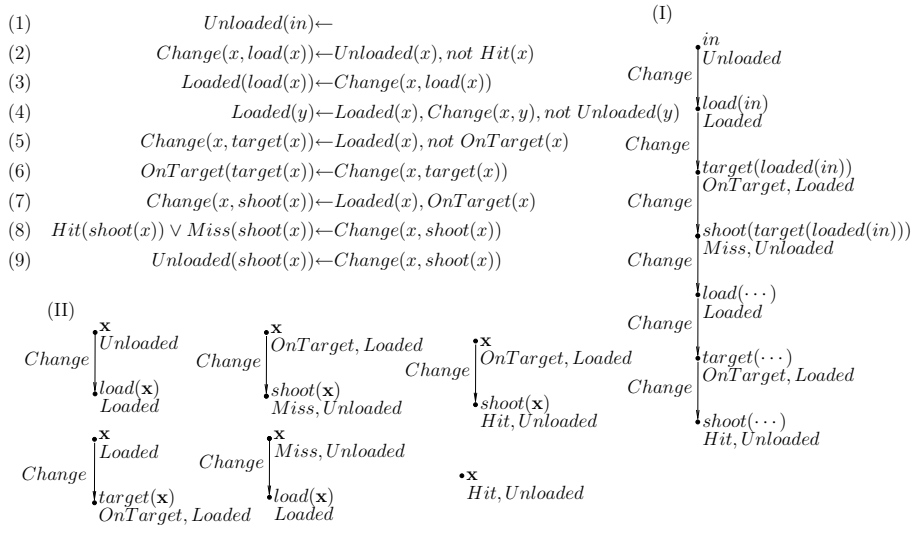


Fig. 2. Example $\mathbb{FDNC}$ program

the target in the second round. The inertia rule (4) demonstrates the capability of $\mathbb{FDNC}$ programs (and ASP, in general) to model common-sense reasoning. Inertia is modeled elegantly as a default that may have exceptions, in terms of direct or indirect effects by some actions that cause a value change.

Inspired by modal logics, decidability of $\mathbb{FDNC}$ programs is ensured via a restricted syntax that ensures the forest model property, combined with certain modularity. More precisely, the first property ensures that each stable model of a program can be viewed as a labeled forest, while the second characterizes global minimality of interpretations in terms of local minimality conditions. The knot technique was exploited to finitely represent such forest-shaped models.

For instance, in the example above, the answer sets of the program can be represented by a coherent knot set depicted in (II) of Figure 2. Here each knot k is a simple tree whose root $\mathbf{x}$ has at most one child $f(\mathbf{x})$, i.e., k is a line, where f is one of the function symbols (*load*, *target*, *shoot*), and each node is labeled with a subset of the fluents (*OnTarget*, *Loaded*, *Unloaded*, *Hit*, *Miss*). Informally, k represents a possible transition from a generic state $\mathbf{x}$ (determined by the associated fluent values) to a successor state by taking the action f. The knot of depth 0 indicates that no action is applicable when the target is hit (more precisely, the state $\{Hit, Unloaded\}$ is reached).[2]

The knot representation of models provides a basis for solving basic reasoning problems associated with ASP, and the technique also allows to infer tight

[2] Note that in this encoding there exists at most one executable action for each state, hence the answer sets are "line-shaped". The situations when several actions are executable would be reflected by branching; the resulting answer sets would be tree-shaped, each branch corresponding to a possible evolution of the initial situation.

complexity bounds. Consistency testing in $\mathbb{FDNC}$ is ExpTime-complete and can be verified by checking for non-emptiness of a complete knot set. The same complexity bounds hold for brave inference of existential queries $\exists x.A(x)$, which, e.g., allow to test for plan existence in planning applications. The knot representation allows to provide model-building services, which are useful in various applications. For example, in planning and diagnosis a complete knot set can be used to build relevant parts of a model corresponding to a possible scenario or a plan. Last but not least, for all the aforementioned tasks, knots provide benefits like modularity and knowledge compilation, as discussed in Section 2.3.

3.2 Knots in Description Logics

Description Logics are a well-established branch of logics for knowledge representation and reasoning, and the premier logic-based formalism for modeling concepts (classes of objects) and roles (binary relations between classes) [2]. They have gained increasing attention in different areas including data and information integration, peer-to-peer data management, and ontology-based data access. In particular, they are of major importance in the Semantic Web, as the standard Web ontologies (OWL) family are based on DLs. The recent use of DLs in a widening range of applications has led to the study of new reasoning problems, among which query answering over DL knowledge bases plays an important role. Several recent papers focus on answering queries over semantically enhanced data schemas expressed by means of DL ontologies, cf. [9,16,17,19,26].

Research has focused on *conjunctive queries (CQs)* and their extensions, which stem from the databases field. A CQ q is of the form $A_1(\boldsymbol{t}_1), \ldots, A_n(\boldsymbol{t}_n)$, where the A_i are predicates (in the context of DLs, concept and role names), and the $\boldsymbol{t}_i$ are matching argument lists of constants and variables. The *entailment problem* for such a query q consists on deciding, given q and a DL KB $\mathcal{K}$, whether there is a homomorphic embedding (informally, a *mapping*) of q into every model of $\mathcal{K}$, i.e., whether $\exists \boldsymbol{x}.A_1(\boldsymbol{t}_1) \wedge \cdots \wedge A_n(\boldsymbol{t}_n)$ is logically implied by $\mathcal{K}$, where $\boldsymbol{x}$ lists all variables in q.[3]

The considered query languages have as a common feature the use of variables to join relations in a flexible way. Such joins are not expressible in the syntax of DLs, hence it is unclear how to easily reduce the query entailment problem to satisfiability testing or any other 'traditional' DL reasoning task. Furthermore, recent complexity results show that for many DLs such reductions are exponential in general. Thus many algorithms rely on machinery beyond that of DLs to traverse sets of models and to test the existence of query mappings in them.

In this setting, knots proved to be a powerful and well-suited reasoning tool. Given a KB $\mathcal{K}$ and a query q, a complete set of knots L can be used to represent a set $\mathbf{M}$ of *canonical models* of $\mathcal{K}$, which can be shown to contain a countermodel M for q whenever q is not entailed by $\mathcal{K}$. Furthermore, it is possible to decide the existence of such a countermodel $M \in \mathbf{M}$ (in time that is polynomial in

[3] More precisely, these are Boolean CQs. More general CQs with distinguished (output) variables and can be easily reduced to Boolean CQs.

L) as follows. The knots in L are enriched with 'markings' that represent a set of subqueries of q. A knot k is marked with a set of subqueries Q, if in every submodel of a model in $\mathbf{M}$ that starts with an instance of k, there is a mapping for some subquery $q \in Q$. The markings can be built inductively considering the depth of these mappings. Simple queries that are entailed at the root of a knot (thus within depth 0) are computed first; in an inductive step, subqueries that are entailed within greater depth are computed by combining the current knot with the markings of its possible successors, until a fixed-point is reached. Let L_q denote the set of knots that are marked with $\{q\}$ at the end of this process. Then there is a query mapping in some $M \in \mathbf{M}$ iff M starts with an instance of a knot in L_q. Thus $\mathbf{M}$ contains a countermodel for q iff there is some knot in $L \setminus L_q$ that can start the model construction.[4]

We give a rough illustration of this procedure on an example. Suppose that a knowledge base $\mathcal{K}$ is given, whose canonical models are represented by the set of knots in Figure 1, part (I), and that they all start with an instance of the knot k_1. We want to decide the entailment of the query

$$q = B(x), P(x, y), C(y), Q(y, z), E(z).$$

We first take the only relevant subquery of it that can be mapped within depth 0, which is $q_1 = E(z)$ (i.e., the query restricted to the variable z; we do not need to consider subqueries where a variable is mapped but its successors are not). We can map it at the root of k_5, this is shown on part (I) of Figure 3. Note that in this figure we have omitted the terms in the knots, and we are using dotted arcs to relate the leaves of each knot to the knots that can follow it in the construction of a model. In the second step, which is illustrated in part (II), we consider a larger subquery $q_2 = C(y), Q(y, z), E(z)$. We add a new marking showing that q_2 can always be mapped starting from the the knot k_4, since its only successor is k_5 which is already marked with $\{q_1\}$. Part (III) shows the third step, where k_1 is used to extend the mapping of q_2 at k_4 to q, exploiting the fact that the right successor of k_1 must be followed by k_4 in every model construction. Given that all models of $\mathcal{K}$ start with an instance of k_1, this already ensures that q is entailed by $\mathcal{K}$. I.e., in every model of $\mathcal{K}$ there is a mapping for q, including, for example, the trees (II) and (III) of Figure 1. A fixed-point is reached after one more iteration that marks k_2 with $\{q\}$ (meaning that, if there were any models starting with an instance of k_2, they would have a mapping for q).

Such a knot-based algorithm for CQ answering in the DL $\mathcal{ALCH}$ was proposed in [28]. Unlike previous algorithms requiring double exponential time, it yields a tight ExpTime upper bound.

The algorithm was extended to $\mathcal{SH}$ in [27]. In this case, however, there are exponentially many relevant subqueries whose combinations may result in different markings. Hence doubly exponentially many (in the combined size of $\mathcal{K}$ and q) knot-marking combinations may be generated in the worst case, resulting in

[4] For simplicity we only consider models that are trees and start at an instance of one knot. This naturally generalizes to forests: for connected q, non-entailment simply requires that there is a set of knots in $L \setminus L_q$ that can start a forest construction.

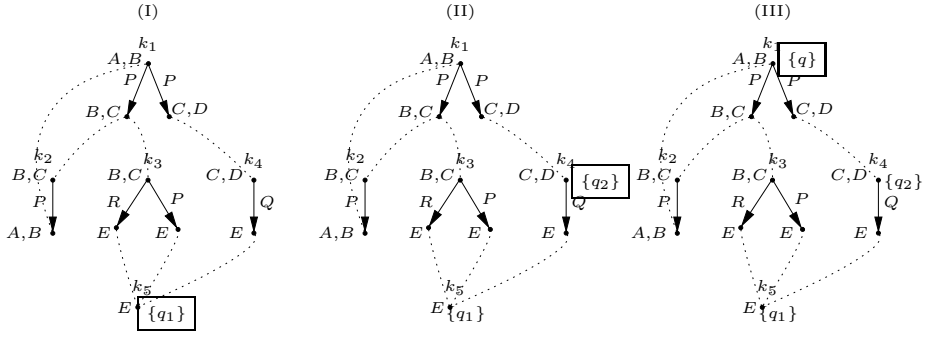


Fig. 3. Query entailment using knots

a 2ExpTime upper bound known from [16,9]. Most recently, this was shown to be tight: CQ answering for $\mathcal{SH}$ is 2ExpTime-hard [13]. Further, a large family of queries for which the $\mathcal{SH}$ algorithm runs in ExpTime has been identified in [27]. This is noticable as in most of the previous query answering algorithms that build on standard techniques for DL reasoning (e.g., automata [9], tableaux [26] and resolution [19]), it was less clear how results that significantly lower the 2ExpTime upper bound for a rich class of queries could be obtained.

A similar technique based on *domino systems* (related to the domino sets of [32]) was used for Horn-$\mathcal{SHIQ}$ in [12]. Like knot sets, domino systems provide a finite representation of models in terms of simple, small building blocks. The main difference is that the domino system represents just one model and there is an explicit transition relation indicating the successors of each node. Such a representation is well suited for Horn-$\mathcal{SHIQ}$, since it is a deterministic DL and query answering can be done over a single *universal model*.

Obtaining new algorithms and complexity bounds is not the only advantage of exploiting knots in the context of query answering over DLs. Indeed, the above mentioned algorithms share other positive features, including the following:

- **Modularity.** The maximal coherent knot set is computed using only the terminological information in the KB. Once this has been done, markings for several queries can be computed over the set, and further queries can be incorporated incrementally (in polynomial time if their size is bounded). All this process is independent of the input data (extensional information).
- **Optimal data complexity.** The set of knots and the query markings can be precomputed in constant time w.r.t. the input data, assuming that the terminological part of the KB and the query are fixed. After this first step, non-entailment of a query can be easily established in non-deterministic polynomial time. Hence the algorithm runs in coNP in *data complexity* and is optimal, as matching hardness is known already for very weak DLs.
- **Datalog encoding.** The precompiled structure consisting of the marked knots can be easily encoded into a Datalog program with unstratified negation (or an equivalent disjunctive Datalog program) that evaluates the query

over any given set of facts. Due to the availability of highly optimized Datalog engines, this seems particularly promising for implementations.

Finally, we point out that this kind of techniques are not only useful for query answering, but even for more traditional reasoning tasks. For example, the authors of [32] fruitfully used their domino sets in an algorithm for testing TBox satisfiability in the DL $\mathcal{SHIQ}$ that can be implemented using OBDDs.

4 Conclusion

As we have briefly discussed above, knots proved to be a useful tool for solving some reasoning tasks in Answer Set Programming and Description Logics. There are several issues that remain to be considered and open interesting avenues for future research.

• **Building knot sets.** A crucial task for knot reasoning is to build coherent knot sets, and in particular the maximal one. Under certain conditions, the maximal set K can be obtained by a simple elimination algorithm that starts with the set of all possible knots (or any superset of K), and removes one by one the knots that cause a violation of the local or the global conditions. Such an elimination is possible, for example, in the applications above, where the global conditions simply require that in a coherent knot set K each leaf of each knot k has a compatible 'successor' in K.[5] However, this method is not efficient in general, particularly when the resulting knot set is small. Furthermore, for particular reasoning problems, it might not be necessary to construct the full set K, but only a relevant part of it on demand (e.g., for query answering if the knowledge base is known to be satisfiable). Hence modular/incremental computation of knot sets is an important issue that remains to be explored.

• **Generalizations.** The knot concept is rather primitive, and for certain applications generalizations are needed. They can be of different kind; one concerns information associated with a knot (in a similar manner as in query answering in DLs), while another concerns structure. Generalizations to subtrees of depth $n \geq 1$ are simple but hardly increase the expressiveness; knots for settings in which models are not tree or forest-shaped would be of higher interest. As pointed out above, our deployment of knots to reasoning tasks in DLs was via related logic program classes, rather than through the link to modal logics and mosaics. Given that ASP also has a link to (non-monotonic) modal logic [23], it remains to be seen whether more general mosaic techniques than knots can be fruitfully exploited via this link.

• **Hybrid techniques.** As several other reasoning techniques had been devised, the question is whether knots can be fruitfully be combined with them. For instance, in query answering one might use the knot technique on a partial knot

[5] More generally, this elimination is applicable whenever the global condition for K can be characterized, by means of a decidable relation $R(k, S)$ between knots k and knot sets S, as follows: for each $k \in K$, there is $S \subseteq K$ such that $R(k, S)$ holds.

set sufficient for singling out the result, and employ other methods for subtasks, such as possible satisfiability tests. However, developing sophisticated hybrid techniques beyond simple combinations is more of a long term goal.

• **Further applications.** Finally, it would be interesting to see further applications of knots, not necessarily in the context of knowledge representation and reasoning. Hybrid knowledge bases might be a first target.

References

1. Andréka, H., Németi, I., van Benthem, J.: Modal languages and bounded fragments of predicate logic. Journal of Philosophical Logic 27(3), 217–274 (1998)
2. Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P.F. (eds.): The Description Logic Handbook: Theory, Implementation and Applications. Cambridge University Press, Cambridge (2003)
3. Baader, F., Sattler, U.: An overview of tableau algorithms for description logics. Studia Logica 69(1), 5–40 (2001)
4. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press, Cambridge (2003)
5. Baselice, S., Bonatti, P.A., Criscuolo, G.: On finitely recursive programs. In: Dahl, V., Niemelä, I. (eds.) ICLP 2007. LNCS, vol. 4670, pp. 89–103. Springer, Heidelberg (2007)
6. Bonatti, P., Baselice, S.: Composing normal programs with function symbols. In: Proc. of ICLP 2008. LNCS. Springer, Heidelberg (to appear, 2008)
7. Calimeri, F., Cozza, S., Ianni, G., Leone, N.: Computable functions in ASP: Theory and implementation. In: Proc. of ICLP 2008. LNCS, vol. 5366. Springer, Heidelberg (2008)
8. Calvanese, D., De Giacomo, G., Lenzerini, M.: Reasoning in expressive description logics with fixpoints based on automata on infinite trees. In: Proc. of the 16th Int. Joint Conf. on Artificial Intelligence (IJCAI 1999), pp. 84–89 (1999)
9. Calvanese, D., Eiter, T., Ortiz, M.: Answering regular path queries in expressive description logics: An automata-theoretic approach. In: Proc. of the 22nd Nat. Conf. on Artificial Intelligence (AAAI 2007), pp. 391–396 (2007)
10. de Bruijn, J., Eiter, T., Polleres, A., Tompits, H.: On representational issues about combinations of classical theories with nonmonotonic rules. In: Lang, J., Lin, F., Wang, J. (eds.) KSEM 2006. LNCS, vol. 4092, pp. 1–22. Springer, Heidelberg (2006)
11. Eiter, T., Faber, W., Leone, N., Pfeifer, G., Polleres, A.: A logic programming approach to knowledge-state planning: Semantics and complexity. ACM Trans. Comput. Log. 5(2), 206–263 (2004)
12. Eiter, T., Gottlob, G., Ortiz, M., Šimkus, M.: Query answering in the description logic Horn-$\mathcal{SHIQ}$. In: Proc. of JELIA 2008. LNCS. Springer, Heidelberg (to appear, 2008)
13. Eiter, T., Lutz, C., Ortiz, M., Šimkus, M.: Complexity of Conjunctive Query Answering in Description Logics with Transitive Roles. Technical report (preliminary), INFSYS RR-1843-08-09, TU Wien (2008)
14. Fox, D., Gomes, C.P. (eds.): Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17. AAAI Press, Menlo Park (2008)
15. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. New Generation Computing 9, 365–385 (1991)

16. Glimm, B., Horrocks, I., Lutz, C., Sattler, U.: Conjunctive query answering for the description logic $\mathcal{SHIQ}$. In: Proc. of IJCAI 2007, pp. 399–404 (2007)
17. Glimm, B., Horrocks, I., Sattler, U.: Conjunctive query entailment for $\mathcal{SHOQ}$. In: Proc. of the 2007 Description Logic Workshop (DL 2007). CEUR Electronic Workshop Proceedings, vol. 250, pp. 65–75 (2007), http://ceur-ws.org/Vol-250/
18. Grädel, E., Kolaitis, P.G., Vardi, M.Y.: On the decision problem for two-variable first-order logic. Bulletin of Symbolic Logic 3(1), 53–69 (1997)
19. Hustadt, U., Motik, B., Sattler, U.: A decomposition rule for decision procedures by resolution-based calculi. In: Baader, F., Voronkov, A. (eds.) LPAR 2004. LNCS, vol. 3452, pp. 21–35. Springer, Heidelberg (2005)
20. Marek, V.W., Remmel, J.B.: On the expressibility of stable logic programming. Theory and Practice of Logic Programming 3, 551–567 (2003)
21. Marek, W., Nerode, A., Remmel, J.: How Complicated is the Set of Stable Models of a Recursive Logic Program? Annals of Pure and Applied Logic 56, 119–135 (1992)
22. Marek, W., Nerode, A., Remmel, J.: The Stable Models of a Predicate Logic Program. Journal of Logic Programming 21(3), 129–153 (1994)
23. Marek, W., Truszczyński, M.: Nonmonotonic Logics – Context-Dependent Reasoning. Springer, Heidelberg (1993)
24. Motik, B., Horrocks, I., Rosati, R., Sattler, U.: Can OWL and logic programming live together happily ever after? In: Cruz, I., Decker, S., Allemang, D., Preist, C., Schwabe, D., Mika, P., Uschold, M., Aroyo, L.M. (eds.) ISWC 2006. LNCS, vol. 4273, pp. 501–514. Springer, Heidelberg (2006)
25. Németi, I.: Free algebras and decidability in algebraic logic. DSc. thesis, Mathematical Institute of The Hungarian Academy of Sciences, Budapest (1986)
26. Ortiz, M., Calvanese, D., Eiter, T.: Data complexity of query answering in expressive description logics via tableaux. J. of Automated Reasoning (June 2008)
27. Ortiz, M., Šimkus, M., Eiter, T.: Conjunctive query answering in $\mathcal{SH}$ using knots. In: Baader, F., Lutz, C., Motik, B. (eds.) Proc. of DL 2008, Dresden, Germany, May 13-16. CEUR Workshop Proceedings, vol. 353 (2008), CEUR-WS.org
28. Ortiz, M., Šimkus, M., Eiter, T.: Worst-case optimal conjunctive query answering for an expressive description logic without inverses. In: Fox and Gomes [14], pp. 504–510
29. Pratt, V.R.: Models of program logics. In: FOCS, pp. 115–122. IEEE, Los Alamitos (1979)
30. Pratt-Hartmann, I.: Complexity of the guarded two-variable fragment with counting quantifiers. J. Log. Comput. 17(1), 133–155 (2007)
31. Rosati, R.: Integrating Ontologies and Rules: Semantic and Computational Issues. In: Barahona, P., Bry, F., Franconi, E., Henze, N., Sattler, U. (eds.) Reasoning Web 2006. LNCS, vol. 4126, pp. 128–151. Springer, Heidelberg (2006)
32. Rudolph, S., Krötzsch, M., Hitzler, P.: Terminological reasoning in $\mathcal{SHIQ}$ with ordered binary decision diagrams. In: Fox and Gomes [14], pp. 529–534
33. Šimkus, M., Eiter, T.: FDNC: Decidable non-monotonic disjunctive logic programs with function symbols. In: Proceedings of LPAR 2007. LNCS, vol. 4790, pp. 514–530. Springer, Heidelberg (2007); Full paper Tech. Rep. INFSYS RR-1843-08-01, TU Vienna, http://www.kr.tuwien.ac.at/research/reports/rr0801.pdf
34. Vardi, M.Y., Wolper, P.: Automata-theoretic techniques for modal logics of programs. J. Comput. Syst. Sci. 32, 183–221 (1986)

Role Conjunctions in Expressive Description Logics

Birte Glimm and Yevgeny Kazakov

Oxford University Computing Laboratory

Abstract. We show that adding role conjunctions to the Description Logics (DLs) $\mathcal{SHI}$ and $\mathcal{SHOIF}$ causes a jump in the computational complexity of the standard reasoning tasks from ExpTime-complete to 2ExpTime-complete and from NExpTime-complete to N2ExpTime-hard respectively. We further show that this increase is due to a subtle interaction between inverse roles, role hierarchies, and role transitivity in the presence of role conjunctions and that for the DL $\mathcal{SHQ}$ a jump in the computational complexity cannot be observed.

1 Introduction

Description Logics are knowledge representation formalisms [1], which are mostly based on decidable fragments of First-Order Logic with only unary and binary predicates, called concepts and roles. The DLs $\mathcal{SHIF}$ and $\mathcal{SHOIN}$ provide a logical underpinning for the W3C standards OWL Lite and OWL DL and highly optimized reasoner implementations are available.

Current standardization efforts go into the direction of also supporting a richer set of constructors for roles. It was recently shown that role compositions in the proposed OWL2 standard cause an exponential blowup [2] in the computational complexity of the standard reasoning problems. We show that allowing for conjunctions over roles can also cause such a blowup.

Role conjunctions are closely related to conjunctive queries. In [3] it was shown how the problem of conjunctive query answering over $\mathcal{SHIQ}$ can be reduced to reasoning in $\mathcal{SHIQ}^{\sqcap}$—the extension of $\mathcal{SHIQ}$ with role conjunctions. For example, the query $\langle x \rangle \leftarrow r(x,y) \wedge s(x,y) \wedge A(y)$ can be answered by retrieving all instances of the concept $\exists(r \sqcap s).A$ for A a concept name, r, s roles, and x, y variables. In [3] it was also shown that reasoning in $\mathcal{SHIQ}^{\sqcap}$ is in 2ExpTime. It was an open question whether this bound is tight.

In this paper we demonstrate that standard reasoning in $\mathcal{SHIQ}^{\sqcap}$ and even in $\mathcal{SHI}^{\sqcap}$ is 2ExpTime-hard. It follows from the construction in [3] that reasoning in $\mathcal{SHIQ}^{\sqcap}$ is in ExpTime when either the number of transitive roles in role inclusions, or the length of role conjunctions is bounded. We also demonstrate that reasoning in $\mathcal{SHIQ}^{\sqcap}$ without inverse roles is in ExpTime as well. Thus, the increased complexity of $\mathcal{SHIQ}^{\sqcap}$ is due to a combination of inverse roles, role transitivity, role hierarchies, and role conjunctions. A similar effect is observed for propositional dynamic logics (PDL), where the intersection operator causes

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 391–405, 2008.

a complexity jump from ExpTime to 2ExpTime [4]. PDL is closely related to the DL $\mathcal{ALC}$ extended with regular expressions on roles.

We now introduce some basic definitions and notations used throughout the paper. In Section 3, we prove that for $\mathcal{SHQ}^{\sqcap}$ the standard reasoning tasks remain in ExpTime. In Section 4, we present the 2ExpTime-hardness result for $\mathcal{SHI}^{\sqcap}$ by a reduction to the word problem for exponentially space bounded Turing machines. In Section 5 we demonstrate that $\mathcal{SHOIF}^{\sqcap}$ is already N2ExpTime-hard using a reduction to domino tiling problems. This paper is accompanied by a technical report which contains intermediate lemmata and full proofs [5].

2 Preliminaries

Let N_C, N_R, and N_I be countably infinite sets of *concept names*, *role names*, and *individual names*. We assume that the set of role names contains a subset $N_{tR} \subseteq N_R$ of *transitive role names*. A *role* R is an element of $N_R \cup \{r^- \mid r \in N_R\}$, where roles of the form r^- are called *inverse roles*. A *role conjunction* is an expression of the form $\rho = (R_1 \sqcap \cdots \sqcap R_n)$. A *role inclusion axiom* (RIA) is an axiom of the form $R \sqsubseteq S$ where R and S are roles. A *role hierarchy* $\mathcal{R}$ is a finite set of role inclusion axioms.

An *interpretation* $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ consists of a non-empty set $\Delta^{\mathcal{I}}$, the *domain* of $\mathcal{I}$, and a function $\cdot^{\mathcal{I}}$, which maps every concept name A to a subset $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$, every role name $r \in N_R$ to a binary relation $r^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$, every role name $r \in N_{tR}$ to a transitive binary relation $r^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$, and every individual name a to an element $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$. The interpretation of an inverse role r^- is $\{\langle d, d' \rangle \mid \langle d', d \rangle \in r^{\mathcal{I}}\}$. The interpretation of a role conjunction $R_1 \sqcap \cdots \sqcap R_n$ is ${R_1}^{\mathcal{I}} \cap \cdots \cap {R_n}^{\mathcal{I}}$. An interpretation $\mathcal{I}$ satisfies a RIA $R \sqsubseteq S$ if $R^{\mathcal{I}} \subseteq S^{\mathcal{I}}$, and a role hierarchy $\mathcal{R}$ if $\mathcal{I}$ satisfies all RIAs in $\mathcal{R}$.

For a role hierarchy $\mathcal{R}$, we introduce the following standard DL notations:

1. We define the function Inv over roles as $\mathsf{Inv}(r) := r^-$ and $\mathsf{Inv}(r^-) := r$ for $r \in N_R$.
2. We define $\sqsubseteq_{\mathcal{R}}$ as the smallest transitive reflexive relation on roles such that $R \sqsubseteq S \in \mathcal{R}$ implies $R \sqsubseteq_{\mathcal{R}} S$ and $\mathsf{Inv}(R) \sqsubseteq_{\mathcal{R}} \mathsf{Inv}(S)$. We write $R \equiv_{\mathcal{R}} S$ if $R \sqsubseteq_{\mathcal{R}} S$ and $S \sqsubseteq_{\mathcal{R}} R$.
3. A role R is called *transitive w.r.t.* $\mathcal{R}$ (notation $R^+ \sqsubseteq_{\mathcal{R}} R$) if $R \equiv_{\mathcal{R}} S$ for some role S such that $S \in N_{tR}$ or $\mathsf{Inv}(S) \in N_{tR}$.
4. A role S is called *simple w.r.t.* $\mathcal{R}$ if there is no role R such that R is transitive w.r.t. $\mathcal{R}$ and $R \sqsubseteq_{\mathcal{R}} S$. A role conjunction $R_1 \sqcap \cdots \sqcap R_n$ is *simple w.r.t.* $\mathcal{R}$ if each conjunct is simple w.r.t. $\mathcal{R}$.

The set of $\mathcal{SHOIQ}^{\sqcap}$*-concepts* is the smallest set built inductively from N_C, N_R, and N_I using the following grammar, where $A \in N_C$, $o \in N_I$, n is a non-negative integer, ρ is a role conjunction and δ is a simple role conjunction:

$$C ::= A \mid \{o\} \mid \neg C \mid C_1 \sqcap C_2 \mid \forall \rho.C \mid \geqslant n\, \delta.C.$$

We use the following standard abbreviations: $C_1 \sqcup C_2 \equiv \neg(\neg C_1 \sqcap \neg C_2)$, $\exists \rho.C \equiv \neg(\forall \rho.(\neg C))$, and $\leqslant n\, \delta.C \equiv \neg(\geqslant (n+1)\, \delta.C)$.

The interpretation of concepts in $\mathcal{I}$ is defined as follows:

$$\{o\}^{\mathcal{I}} = \{o^{\mathcal{I}}\},\ (C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}},\ (\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}},$$
$$(\forall \rho.C)^{\mathcal{I}} = \{d \in \Delta^{\mathcal{I}} \mid \text{ if } \langle d, d' \rangle \in \rho^{\mathcal{I}}, \text{ then } d' \in C^{\mathcal{I}}\},$$
$$(\geqslant n\, \delta.C)^{\mathcal{I}} = \{d \in \Delta^{\mathcal{I}} \mid \sharp s^{\mathcal{I}}(d, C) \geq n\}$$

where $\sharp M$ denotes the cardinality of the set M and $s^{\mathcal{I}}(d, C)$ is defined as $\{d' \in \Delta^{\mathcal{I}} \mid \langle d, d' \rangle \in s^{\mathcal{I}}$ and $d' \in C^{\mathcal{I}}\}$. Concepts of the form $\{o\}$ are called *nominals*.

The DL $\mathcal{SHOIF}^{\sqcap}$ is obtained by only allowing for the declaration of roles as functional (e.g., $\mathsf{Func}(R)$) instead of full number restrictions. By disallowing number restrictions and nominals, we obtain $\mathcal{SHI}^{\sqcap}$. Finally, $\mathcal{SHOIQ}^{\sqcap}$ minus nominals and inverse roles, results in the DL $\mathcal{SHQ}^{\sqcap}$.

A *general concept inclusion* (GCI) is an expression $C \sqsubseteq D$, where both C and D are concepts. A finite set of GCIs is called a *TBox*. An interpretation $\mathcal{I}$ *satisfies a GCI* $C \sqsubseteq D$ if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$, and a TBox $\mathcal{T}$ if it satisfies every GCI in $\mathcal{T}$.

An (ABox) *assertion* is an expression of the form $C(a)$, $r(a, b)$, where C is a concept, r a role, and $a, b \in N_I$. An *ABox* is a finite set of assertions. We use $N_I(\mathcal{A})$ to denote the set of individual names occurring in $\mathcal{A}$. An interpretation $\mathcal{I}$ *satisfies an assertion* $C(a)$ if $a^{\mathcal{I}} \in C^{\mathcal{I}}$, $r(a, b)$ if $\langle a^{\mathcal{I}}, b^{\mathcal{I}} \rangle \in r^{\mathcal{I}}$. An interpretation $\mathcal{I}$ *satisfies an ABox* $\mathcal{A}$ if it satisfies each assertion in $\mathcal{A}$, denoted as $\mathcal{I} \models \mathcal{A}$.

A *knowledge base* (KB) is a triple $(\mathcal{R}, \mathcal{T}, \mathcal{A})$ with $\mathcal{R}$ a role hierarchy, $\mathcal{T}$ a TBox, and $\mathcal{A}$ an ABox. Let $\mathcal{K} = (\mathcal{R}, \mathcal{T}, \mathcal{A})$ be a knowledge base and $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ an interpretation. We say that $\mathcal{I}$ *satisfies* $\mathcal{K}$ if $\mathcal{I}$ satisfies $\mathcal{R}$, $\mathcal{T}$, and $\mathcal{A}$. In this case, we say that $\mathcal{I}$ is a *model* of $\mathcal{K}$ and write $\mathcal{I} \models \mathcal{K}$. We say that $\mathcal{K}$ *is satisfiable* if $\mathcal{K}$ has a model. A concept D *subsumes* a concept C w.r.t. $\mathcal{K}$ if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ for every model $\mathcal{I}$ of $\mathcal{K}$. A concept C is *satisfiable* w.r.t. $\mathcal{K}$ if there is a model $\mathcal{I}$ of $\mathcal{K}$ such that $C^{\mathcal{I}} \neq \emptyset$. It should be noted that the standard reasoning tasks such as knowledge base satisfiability, concept subsumption, or concept satisfiability are mutually reducible in polynomial time.

3 $\mathcal{SHQ}^{\sqcap}$ Is ExpTime-Complete

In this section, we show that adding role conjunctions to the DL $\mathcal{SHQ}$ does not increase the computational complexity of the standard reasoning tasks. For this purpose, we devise a polynomial translation of a given $\mathcal{SHQ}^{\sqcap}$ knowledge base to an equisatisfiable $\mathcal{ALCHQ}^{\sqcap}$ knowledge base (i.e., $\mathcal{SHQ}^{\sqcap}$ minus role transitivity) for which the standard reasoning tasks are ExpTime-complete [6,3,7].

Let $\mathcal{K} = (\mathcal{R}, \mathcal{T}, \mathcal{A})$ be an $\mathcal{SHQ}^{\sqcap}$ knowledge base. We say that $\mathcal{K}$ is *simplified* if $\mathcal{T}$ contains only axioms of the form:

$$A \sqsubseteq \forall \rho.B \ \mid\ A \sqsubseteq \exists \rho.B \ \mid\ A \sqsubseteq \bowtie n\, \delta.B \ \mid\ \bigsqcap A_i \sqsubseteq \bigsqcup B_j,$$

where $A_{(i)}$ and $B_{(j)}$ are atomic concepts, ρ (δ) is a (simple) conjunction of roles, and $\bowtie$ stands for $\leqslant$ or $\geqslant$. Furthermore, concept assertions in $\mathcal{A}$ are limited to the form $A(a)$ for A a concept name. Every $\mathcal{SHQ}^{\sqcap}$ knowledge base, which is not in this form, can be transformed in polynomial time into the desired form by using

the standard structural transformation, which iteratively introduces definitions for compound sub-concepts and sub-roles (see, e.g., [8]).

It is well known that transitivity can be eliminated from $\mathcal{SHIQ}$ and $\mathcal{SHOIQ}$ knowledge bases by using auxiliary axioms that propagate the concepts over transitive roles [8,9]. The transitivity elimination has been extended to $\mathcal{SHIQ}^{\sqcap}$ [3], however it becomes exponential in the worst case, since one has to introduce new axioms for (possibly exponentially many) conjunctions of transitive roles. The procedure is, however, polynomial if either the number of transitive roles in role inclusions or the length of role conjunctions is bounded. Below we describe a polynomial elimination of transitivity when there are no role inverses.

Let $\mathcal{K} = (\mathcal{R}, \mathcal{T}, \mathcal{A})$ be a simplified $\mathcal{SHQ}^{\sqcap}$ knowledge base. We construct an $\mathcal{ALCHQ}^{\sqcap}$ knowledge base $\mathcal{K}' = (\mathcal{R}', \mathcal{T}', \mathcal{A}')$ from $\mathcal{K}$ as follows. The signature of $\mathcal{K}'$ is defined by $N_I(\mathcal{K}') := N_I(\mathcal{K})$, $N_R(\mathcal{K}') := N_R(\mathcal{K})$, $N_{tR}(\mathcal{K}') := \emptyset$, $N_C(\mathcal{K}') := N_C(\mathcal{K}) \cup \{A_a, A_a^r \mid A \in N_C(\mathcal{K}), a \in N_I(\mathcal{K}), r \in N_R(\mathcal{K})\}$. Recall, that w.l.o.g., $N_I(\mathcal{K})$ is non-empty, therefore there exists at least one A_a for every $A \in N_C(\mathcal{K})$. We set $\mathcal{R}' := \mathcal{R}$, $\mathcal{A}' := \mathcal{A}$, and $\mathcal{T}'$ as an extension of $\mathcal{T}$ with the following axioms:

$$A \sqsubseteq \bigsqcup_{a \in N_I(\mathcal{A})} A_a \qquad A \in N_C(\mathcal{K}) \quad (1)$$

$$A_a \sqsubseteq \forall r.A_a^r \qquad A \in N_C(\mathcal{K}),\ a \in N_I(\mathcal{A}),\ r \in N_R(\mathcal{K}) \quad (2)$$

$$A_a^t \sqsubseteq \forall t.A_a^t \qquad A \in N_C(\mathcal{K}),\ a \in N_I(\mathcal{A}),\ t \in N_{tR}(\mathcal{K}) \quad (3)$$

$$A_a^t \sqsubseteq A_a^r \qquad A \in N_C(\mathcal{K}),\ a \in N_I(\mathcal{A}),\ t \in N_{tR}(\mathcal{K}),\ r \in N_R(\mathcal{K}),\ t \sqsubseteq_{\mathcal{R}} r \quad (4)$$

$$A_a^{r_1} \sqcap \cdots \sqcap A_a^{r_n} \sqsubseteq B \qquad a \in N_I(\mathcal{A}),\ (A \sqsubseteq \forall \rho.B) \in \mathcal{T},\ \rho = r_1 \sqcap \cdots \sqcap r_n \quad (5)$$

Theorem 1. *Let $\mathcal{K} = (\mathcal{R}, \mathcal{T}, \mathcal{A})$ be a simplified $\mathcal{SHQ}^{\sqcap}$ knowledge base and $\mathcal{K}' = (\mathcal{R}', \mathcal{T}', \mathcal{A}')$ an $\mathcal{ALCHQ}^{\sqcap}$ knowledge base obtained from $\mathcal{K}$ as described above. Then (i) $\mathcal{K}'$ is obtained from $\mathcal{K}$ in polynomial time and (ii) $\mathcal{K}$ is satisfiable iff $\mathcal{K}'$ is satisfiable.*

Proof (Sketch). Claim (i) is straightforward. We sketch the proof for Claim (ii).

For the "if" direction of (ii), one can show that every model $\mathcal{J}$ of $\mathcal{K}'$ can be extended to a model $\mathcal{I}$ of $\mathcal{K}$ by interpreting non-simple roles $r \in N_R$ as $r^{\mathcal{J}} \cup \bigcup_{t \sqsubseteq_{\mathcal{R}} r,\, t \in N_{tR}} (t^{\mathcal{J}})^+$ and leaving the interpretation of the other symbols unchanged. All axioms that do not have negative occurrences of non-simple roles remain satisfied in $\mathcal{I}$. Among the remaining axioms are RIAs $r \sqsubseteq s$ and axioms of the form $A \sqsubseteq \forall \rho.B$. RIAs $r \sqsubseteq s$ are satisfied by definition of $\mathcal{I}$, and axioms of the form $A \sqsubseteq \forall \rho.B$ are satisfied due to axioms (1)–(5).

For the "only if" direction of (ii) we use the fact that every satisfiable $\mathcal{SHQ}^{\sqcap}$ knowledge base $\mathcal{K}$ has a forest-shaped model $\mathcal{I}$, where the ABox individuals form the roots of the trees and relations can only be between the individuals or within the trees. The model $\mathcal{I}$ can be then extended to a model $\mathcal{J}$ of the axioms (1)–(5) by interpreting A_a as the restriction of A to the elements of the tree growing from a, and A_a^r as the minimal sets satisfying axioms (2)–(4). For proving that $\mathcal{J}$ satisfies all axioms of the form (5), we use a property that if two elements of a tree have a common descendant, then one is a descendant of the other. □

Corollary 1. *The problem of concept satisfiability in the DL $\mathcal{SHQ}^{\sqcap}$ is ExpTime-complete (and so are all the standard reasoning problems).*

4 $\mathcal{SHI}^{\sqcap}$ Is 2ExpTime-Complete

In this section, we show that extending $\mathcal{SHI}$ with role conjunctions causes an exponential blow-up in the computational complexity of the standard reasoning tasks. We show this by a reduction from the word problem of an exponential space alternating Turing machine.

An *alternating Turning machine* (ATM) is a tuple $M = (\Gamma, \Sigma, Q, q_0, \delta_1, \delta_2)$, where Γ is a finite *working alphabet* containing a *blank symbol* $\boxdot$, $\Sigma \subseteq \Gamma \setminus \{\boxdot\}$ is the *input alphabet*; $Q = Q_\exists \uplus Q_\forall \uplus \{q_a\} \uplus \{q_r\}$ is a finite set of states partitioned into *existential states* $Q_\exists$, *universal states* $Q_\forall$, an *accepting state* q_a, and a *rejecting state* q_r; $q_0 \in Q_\exists$ is the *starting state*, and $\delta_1, \delta_2 \colon (Q_\exists \cup Q_\forall) \times \Gamma \to Q \times \Gamma \times \{L, R\}$ are *transition functions*. A *configuration* of M is a word $c = w_1 q w_2$ where $w_1, w_2 \in \Gamma^*$ and $q \in Q$. An *initial configuration* is $c^0 = q_0 w_0$ where $w_0 \in \Sigma^*$. The *size* $|c|$ of a configuration c is the number of symbols in c. The *successor configurations* $\delta_1(c)$ and $\delta_2(c)$ of a configuration $c = w_1 q w_2$ with $q \neq q_a, q_r$ over the transition functions δ_1 and δ_2 are defined as for deterministic Turing machines (see, e.g., [10]). The sets $\mathsf{C}_{\mathsf{acc}}(M)$ of *accepting configurations* and $\mathsf{C}_{\mathsf{rej}}(M)$ of *rejecting configurations* of M are the smallest sets such that (i) $c = w_1 q w_2 \in \mathsf{C}_{\mathsf{acc}}(M)$ if either $q = q_a$, or $q \in Q_\forall$ and $\delta_1(c), \delta_2(c) \in \mathsf{C}_{\mathsf{acc}}(M)$, or $q \in Q_\exists$ and $\delta_1(c) \in \mathsf{C}_{\mathsf{acc}}(M)$ or $\delta_2(c) \in \mathsf{C}_{\mathsf{acc}}(M)$, and (ii) $c = w_1 q w_2 \in \mathsf{C}_{\mathsf{rej}}(M)$ if either $q = q_r$, or $q \in Q_\exists$ and $\delta_1(c), \delta_2(c) \in \mathsf{C}_{\mathsf{rej}}(M)$, or $q \in Q_\forall$ and $\delta_1(c) \in \mathsf{C}_{\mathsf{rej}}(M)$ or $\delta_2(c) \in \mathsf{C}_{\mathsf{rej}}(M)$. The set of *reachable* configurations from an initial configuration c^0 in M is the smallest set $M(c^0)$ such that $c^0 \in M(c^0)$ and $\delta_1(c), \delta_2(c) \in M(c^0)$ for every $c \in M(c^0)$. A *word problem* for an ATM M is to decide given an initial configuration c^0 whether $c^0 \in \mathsf{C}_{\mathsf{acc}}(M)$. M is $g(n)$ *space bounded* if for every initial configuration c^0 we have: (i) $c^0 \in \mathsf{C}_{\mathsf{acc}}(M) \cup \mathsf{C}_{\mathsf{rej}}(M)$, and (ii) $|c| \leq g(|c^0|)$ for every $c \in M(c^0)$. A classical result $\mathsf{AExpSpace} = \mathsf{2ExpTime}$ [11] implies that there exists a 2^n space bounded ATM M for which the following decision problem is 2ExpTime-complete: given an initial configuration c^0 decide whether $c^0 \in \mathsf{C}_{\mathsf{acc}}(M)$.

We encode a computation of the ATM M in a binary tree (see Figure 1) whereby the configurations of M are encoded on exponentially long chains that grow from the nodes of the tree—the i^{th} element of a chain represents the i^{th} element of the configuration. In our construction, we distinguish odd and even configurations in the computation using concept names Odd and Even. Every odd configuration has two even successor configurations reachable by roles r_e^1 and r_e^2 respectively; likewise, every even configuration has two odd successor configurations reachable by inverses of r_o^1 and r_o^2. We further alternate between the concepts P_0, P_1, and P_2 within the levels of the binary tree. This allows us to distinguish the predecessor and the successor configuration represented by the exponentially long chains. We enforce these chains (see Figure 2) by using the well know "integer counting" technique [12]. A counter $c^{\mathcal{I}}(x)$ is an integer between 0 and $2^n - 1$ that is assigned to an element x of the interpretation $\mathcal{I}$ using n atomic concepts $B_1, \ldots, B_n$ such that the i^{th} bit of $c^{\mathcal{I}}(x)$ is equal to 1 iff $x \in {B_i}^{\mathcal{I}}$. We first define the concept Z that can be used to initialize the counter to zero, and the concept E to detect whether the counter has reached the final

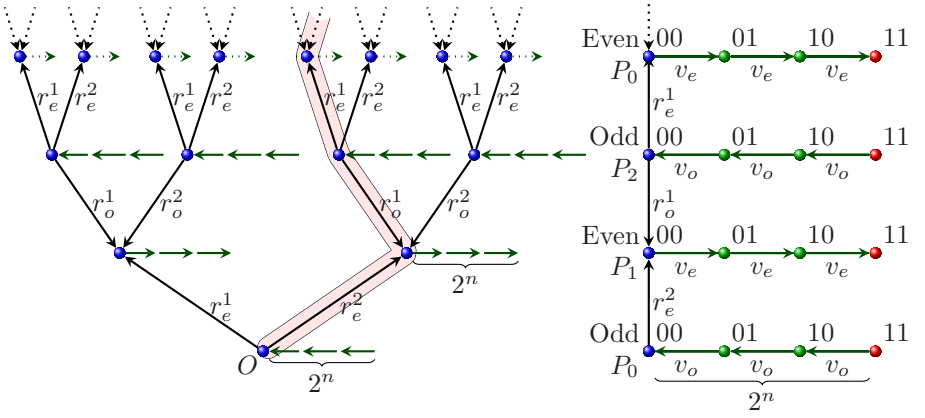


Fig. 1. The alternating binary tree structure for simulating a computation of the ATM (left) and a detailed picture for the highlighted path (right)

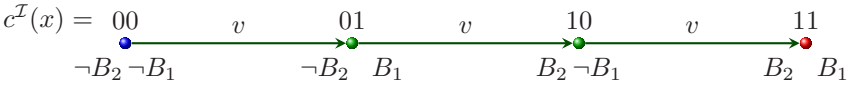


Fig. 2. Expressing exponentially long chains using a counter and binary encoding

value $2^n - 1$ and, thus, the end of the chain is reached:

$$Z \equiv \neg B_1 \sqcap \ldots \sqcap \neg B_n \qquad\qquad E \equiv \quad B_1 \sqcap \ldots \sqcap B_n \tag{6}$$

Every element that is not the end of the chain has a v-successor:

$$\neg E \sqsubseteq \exists v.\top \tag{7}$$

The lowest bit of the counter is always flipped over v, while any other bit of the counter is flipped over v if and only if the previous bit is flipped from 1 to 0:

$$\top \equiv (B_1 \sqcap \forall v.\neg B_1) \sqcup (\neg B_1 \sqcap \forall v.B_1) \tag{8}$$

$$B_{k-1} \sqcap \forall v.\neg B_{k-1} \equiv (B_k \sqcap \forall v.\neg B_k) \sqcup (\neg B_k \sqcap \forall v.B_k) \qquad 1 < k \leq n \tag{9}$$

For convenience, let us denote by $j[i]_2$ the i^{th} bit of j in binary coding (the lowest bit of j is $j[1]_2$).

The tree-like structure in Figure 1 is induced by the following formulas. First, we initialize the origin O of the tree by saying that it belongs to an odd row labeled with P_0 and, with the concept Z, we initialize an exponential chain:

$$O \sqsubseteq \mathrm{Odd} \sqcap P_0 \sqcap Z \tag{10}$$

Every initial element of an exponential chain has two successors alternating between odd and even values:

$$Z \sqcap \mathrm{Odd} \sqsubseteq \exists r_e^1.\mathrm{Even} \sqcap \exists r_e^2.\mathrm{Even} \tag{11}$$

$$Z \sqcap \mathrm{Even} \sqsubseteq \exists r_o^{1^-}.\mathrm{Odd} \sqcap \exists r_o^{2^-}.\mathrm{Odd} \tag{12}$$

For convenience, we introduce super-roles r^1, r^2 and r of the created roles to keep track of the relations between the nodes and their successors:

$$r_e^1 \sqsubseteq r^1 \qquad r_o^1 \sqsubseteq r^{1^-} \qquad r_e^2 \sqsubseteq r^2 \qquad r_o^2 \sqsubseteq r^{2^-} \qquad r^1 \sqsubseteq r \qquad r^2 \sqsubseteq r \tag{13}$$

The new roles are used to initialize the value Z for the successors and increment P_j over r modulo 3 (we denote $j+1 \mod 3$ as $[j+1]_3$):

$$Z \sqsubseteq \forall r.Z \qquad P_j \sqsubseteq \forall r.P_{[j+1]_3} \qquad 0 \le j \le 2 \tag{14}$$

In order to have the roles on the exponential chain correspond to the odd and even rows, we replace axiom (7) with the following axioms:

$$\neg E \sqcap \mathrm{Even} \sqsubseteq \exists v_e.\top \qquad \neg E \sqcap \mathrm{Odd} \sqsubseteq \exists v_o^-.\top \tag{15}$$

$$v_o \sqsubseteq v^- \qquad v_e \sqsubseteq v \tag{16}$$

$$\mathrm{Odd} \sqsubseteq \forall v.\mathrm{Odd} \qquad \mathrm{Even} \sqsubseteq \forall v.\mathrm{Even} \tag{17}$$

The values of P_j are copied across the elements of the same row:

$$P_j \sqsubseteq \forall v.P_j \qquad \neg P_j \sqsubseteq \forall v.\neg P_j \qquad 0 \le j \le 2 \tag{18}$$

If we take a look at Figure 1, we notice that the roles r_o^i, r_e^i, v_o and v_e are directed in such a way that, from every element of an exponential chain, only elements of the neighboring chains are reachable by a sequence of roles. In other words, if we introduce a common transitive super-role t of these roles, then every element of the chain will be connected via t to exactly all elements of the parent chain and all elements of the successor chains. Unfortunately, this is not sufficient to simulate a computation of the Turing machine, as we need to connect exactly the corresponding elements of a chain and its two successor chains to compute the successor configurations. In order to achieve this goal, we will add auxiliary chains to the exponential chain that, using transitive super-roles and role conjunctions, will allow us to restrict the reachability relation only to the corresponding elements.

The detailed construction for the side chains of two successive configurations is shown in Figure 3. Every element of the exponential v-chain has n additional "side" successors reachable by roles h_{ke}^j and h_{ko}^j with $j \in \{0,1\}$ and $1 \le k \le n$. Intuitively, k corresponds to the counting concepts and j to the counter value. We will also count the level in the h-chains using concepts H_k, $0 \le k \le n$—all elements of the v-chain belong to H_0, and every h-successor of an element in H_{k-1} belongs to H_k. The following axioms initialize the side chains according to this description:

$$O \sqsubseteq H_0 \qquad H_0 \sqsubseteq \forall r.H_0 \qquad H_0 \sqsubseteq \forall v.H_0 \tag{19}$$

$$H_{k-1} \sqcap \neg B_k \sqsubseteq (\neg\mathrm{Even} \sqcup \exists h_{ke}^0.H_k) \sqcap (\neg\mathrm{Odd} \sqcup \exists {h_{ko}^0}^-.H_k) \qquad 1 \le k \le n \tag{20}$$

$$H_{k-1} \sqcap B_k \sqsubseteq (\neg\mathrm{Even} \sqcup \exists h_{ke}^1.H_k) \sqcap (\neg\mathrm{Odd} \sqcup \exists {h_{ko}^1}^-.H_k) \qquad 1 \le k \le n \tag{21}$$

$$h_{ke}^j \sqsubseteq h \qquad h_{ko}^j \sqsubseteq h^- \qquad j \in \{0,1\},\ 1 \le k \le n \tag{22}$$

$$\mathrm{Even} \sqsubseteq \forall h.\mathrm{Even} \qquad \mathrm{Odd} \sqsubseteq \forall h.\mathrm{Odd} \tag{23}$$

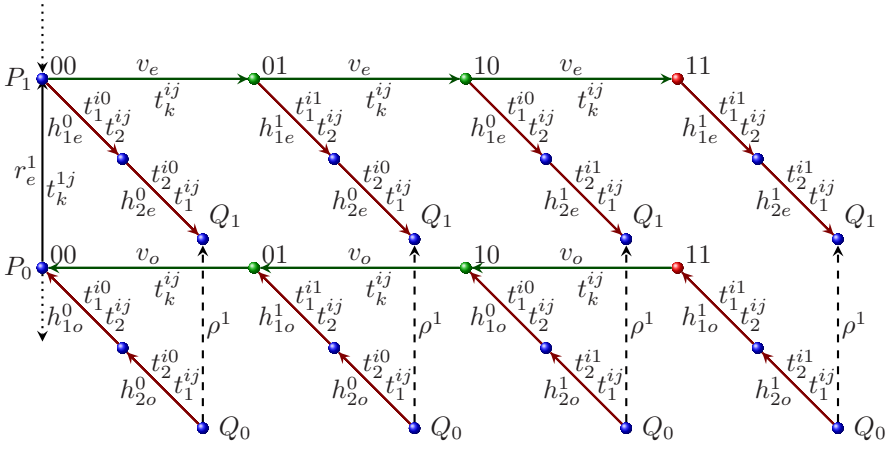


Fig. 3. A zoom-in and extension of Figure 1, which illustrates the use of the auxiliary side chains to connect the elements of the exponentially long chains with the corresponding elements in the successor chains

We use these roles to express that the elements within an h-chain have the same values for B_k and P_j:

$$B_k \sqsubseteq \forall h.B_k \qquad \neg B_k \sqsubseteq \forall h.\neg B_k \qquad 0 \le k \le n \tag{24}$$

$$P_j \sqsubseteq \forall h.P_j \qquad \neg P_j \sqsubseteq \forall h.\neg P_j \qquad 0 \le j \le 2 \tag{25}$$

For the final elements of the h-chains, we introduce the special concepts Q_i that correlate with the concepts P_j:

$$H_n \sqsubseteq (P_j \sqcap Q_j) \sqcup (\neg P_j \sqcap \neg Q_j) \qquad 0 \le j \le 2 \tag{26}$$

These concepts will be used to connect the last elements of the h-chains with the corresponding elements in the chains for the two successor configurations using role conjunctions ρ^1 and ρ^2 introduced later on (see Figure 3). In order to connect these elements, we introduce transitive super-roles t_k^{ij} with $i \in \{1,2\}$, $j \in \{0,1\}$, and $1 \le k \le n$:

$$r_o^i \sqsubseteq t_k^{ij} \qquad r_e^i \sqsubseteq t_k^{ij} \tag{27}$$

$$v_o \sqsubseteq t_k^{ij} \qquad v_e \sqsubseteq t_k^{ij} \tag{28}$$

$$h_{ko}^j \sqsubseteq t_k^{ij} \qquad h_{ke}^j \sqsubseteq t_k^{ij} \tag{29}$$

$$h_{ko}^j \sqsubseteq t_{k'}^{ij'} \qquad h_{ke}^j \sqsubseteq t_{k'}^{ij'} \qquad j' \in \{0,1\},\ 1 \le k' \le n,\ k' \ne k \tag{30}$$

Intuitively, the index i in t_k^{ij} is inherited from the roles r_o^i and r_e^i (27)—all role implications hold for both values of i. Likewise, the index j is inherited from h_{ko}^j and h_{ke}^j, but only when the values of the index k match (29)—otherwise the role implications hold for both values of j (30). Roles v_o and v_e do not filter any indexes and imply all roles t_k^{ij} (28). Axioms (27)–(30) make sure that the

first and the last elements of every h-chain are connected with t_k^{i0} (t_k^{i1}) iff the k^{th} bit of the counter is 0 (1). Thus, only the corresponding last elements of the h-chains in the successor configurations are connected with t_k^{ij} for all k with $1 \leq k \leq n$ and some i and j, because they have the same values for the counter. To make use of this property we introduce roles s_k^i that are obtained from t_k^{ij} by abstracting from j and forgetting the direction:

$$t_k^{ij} \sqsubseteq s_k^i \qquad t_k^{ij^-} \sqsubseteq s_k^i \qquad i \in \{1,2\},\, j \in \{0,1\},\, 1 \leq k \leq n \tag{31}$$

Now define the role conjunctions $\rho^1 = s_1^1 \sqcap \cdots \sqcap s_n^1$ and $\rho^2 = s_1^2 \sqcap \cdots \sqcap s_n^2$ that connect the last elements of the h-chains iff they are the corresponding elements for the r^1 and r^2 successors in our binary tree on Figure 1. Note that ρ^1 and ρ^2 are not simple.

We now specify how the created tree structure relates to an alternating Turing machine. Let c^0 be an initial configuration of an ATM $M = (\Gamma, \Sigma, Q, q_0, \delta_1, \delta_2)$ and $n = |c^0|$ (w.l.o.g., we assume that $n > 2$). In order to decide whether $c^0 \in \mathsf{C}_{\mathsf{acc}}(M)$, we try to build all the required accepting successor configurations of c^0 for M. We encode the configurations of M on the 2^n-long v-chains. A chain corresponding to a configuration c is connected via the roles r^1 and r^2 to two chains that correspond to $\delta_1(c)$ and $\delta_2(c)$ respectively. We use an atomic concept A_a for every symbol a that can occur in configurations and we make sure that all elements of the same h-chain are assigned to the same symbol:

$$A_a \sqsubseteq \forall h.A_a \qquad \neg A_a \sqsubseteq \forall h.\neg A_a \tag{32}$$

It is a well-known property of the transition functions of Turing machines that the symbols c_i^1 and c_i^2 at the position i of $\delta_1(c)$ and $\delta_2(c)$ are uniquely determined by the symbols c_{i-1}, c_i, c_{i+1}, and c_{i+2} of c at the positions $i-1, i, i+1$, and $i+2$.[1] We assume that this correspondence is given by the (partial) functions λ_1 and λ_2 such that $\lambda_1(c_{i-1}, c_i, c_{i+1}, c_{i+2}) = c_i^1$ and $\lambda_2(c_{i-1}, c_i, c_{i+1}, c_{i+2}) = c_i^2$. We use this property in our encoding as follows: for every quadruple of symbols $a_1, a_2, a_3, a_4 \in Q \cup \Gamma$, we introduce a concept name $S_{a_1a_2a_3a_4}$ which expresses that the current element of the v-chain is assigned with the symbol a_2, its v-predecessor with a_1 and its next two v-successors with respectively a_3 and a_4 (a_1, a_3, and a_4 are $\boxdot$ if there are no such elements):

$$Z \sqcap A_{a_2} \sqcap \exists v.(A_{a_3} \sqcap \exists v.A_{a_4}) \sqsubseteq S_{\boxdot a_2 a_3 a_4} \qquad a_2, a_3, a_4 \in Q \cup \Gamma \tag{33}$$

$$A_{a_1} \sqcap \exists v.(A_{a_2} \sqcap \exists v.(A_{a_3} \sqcap \exists v.A_{a_4})) \sqsubseteq \forall v.S_{a_1a_2a_3a_4} \qquad a_1, a_2, a_3, a_4 \in Q \cup \Gamma \tag{34}$$

$$A_{a_1} \sqcap \exists v.(A_{a_2} \sqcap \exists v.(A_{a_3} \sqcap E)) \sqsubseteq \forall v.S_{a_1a_2a_3\boxdot} \qquad a_1, a_2, a_3 \in Q \cup \Gamma \tag{35}$$

$$A_{a_1} \sqcap \exists v.(A_{a_2} \sqcap E) \sqsubseteq \forall v.S_{a_1a_2\boxdot\boxdot} \qquad a_1, a_2 \in Q \cup \Gamma \tag{36}$$

Furthermore, all elements of the same h-chain have the same values of $S_{a_1a_2a_3a_4}$:

$$S_{a_1a_2a_3a_4} \sqsubseteq \forall h.S_{a_1a_2a_3a_4} \qquad \neg S_{a_1a_2a_3a_4} \sqsubseteq \forall h.\neg S_{a_1a_2a_3a_4} \tag{37}$$

[1] If any of the indexes $i-1, i+1$, or $i+2$ are out of range for the configuration c, we assume that the corresponding symbols c_{i-1}, c_{i+1}, and c_{i+2} are the blank symbol $\boxdot$.

Finally, the properties of the transition functions are expressed using the following axioms, where, as previously defined $\rho^1 = s_1^1 \sqcap \dots \sqcap s_n^1$ and $\rho^2 = s_1^2 \sqcap \dots \sqcap s_n^2$:

$$S_{a_1a_2a_3a_4} \sqcap Q_j \sqsubseteq \forall \rho^1.[\neg Q_{[j+1]_3} \sqcup A_{\lambda_1(a_1,a_2,a_3,a_4)}] \qquad 0 \leq i \leq 2 \tag{38}$$

$$S_{a_1a_2a_3a_4} \sqcap Q_j \sqsubseteq \forall \rho^2.[\neg Q_{[j+1]_3} \sqcup A_{\lambda_2(a_1,a_2,a_3,a_4)}] \qquad 0 \leq i \leq 2 \tag{39}$$

Intuitively, these axioms say that whenever $S_{a_1a_2a_3a_4}$ holds at the end of an h-chain where Q_j holds, then $A_{\lambda_1(a_1,a_2,a_3,a_4)}$ should hold for every ρ^1 (ρ^2) successor for which $Q_{[j+1]_3}$ holds. As noted before, only the corresponding last elements of the h-chains can be connected by ρ^1 and ρ^2. The concepts Q_j and $Q_{[j+1]_3}$ restrict the attention to the last elements of the h-chains and make sure that the information is propagated to the successor configuration and not to the predecessor configuration.

We now make sure that the elements in the root chain of our tree correspond to the initial configuration c^0:

$$O \sqsubseteq A_{c_1^0} \sqcap \forall v.(A_{c_2^0} \sqcap \dots \forall v.(A_{c_n^0} \sqcap \forall v.O_\square) \dots) \tag{40}$$

$$O_\square \sqsubseteq A_\square \sqcap \forall v.O_\square \tag{41}$$

In order to distinguish between the configurations with existential and universal states, we introduce two concepts $S_\forall$ and $S_\exists$, which are implied by the corresponding states and propagated to the first elements of the configuration:

$$A_q \sqsubseteq S_\exists \qquad q \in Q_\exists \qquad\qquad A_q \sqsubseteq S_\forall \qquad q \in Q_\forall \tag{42}$$

$$\exists v.S_\exists \sqsubseteq S_\exists \qquad\qquad \exists v.S_\forall \sqsubseteq S_\forall \tag{43}$$

Now instead of always creating two successor configurations, we create only configurations that are required for acceptance. Thus, we replace axioms (11) and (12) with the axioms (44)–(46) below:

$$Z \sqcap \mathrm{Odd} \sqcap S_\forall \sqsubseteq \exists r_e^1.\top \sqcap \exists r_e^2.\top \qquad Z \sqcap \mathrm{Even} \sqcap S_\forall \sqsubseteq \exists r_o^{1^-}.\top \sqcap \exists r_o^{2^-}.\top \tag{44}$$

$$Z \sqcap \mathrm{Odd} \sqcap S_\exists \sqsubseteq \exists r_e^1.\top \sqcup \exists r_e^2.\top \qquad Z \sqcap \mathrm{Even} \sqcap S_\exists \sqsubseteq \exists r_o^{1^-}.\top \sqcup \exists r_o^{2^-}.\top \tag{45}$$

$$\mathrm{Odd} \sqsubseteq \forall r.\mathrm{Even} \qquad\qquad \mathrm{Even} \sqsubseteq \forall r.\mathrm{Odd} \tag{46}$$

Finally we forbid configurations with rejecting states in our model:

$$A_{q_r} \sqsubseteq \bot \tag{47}$$

To summarize, our construction proves the following theorem:

Theorem 2. *Let c^0 be an initial configuration for the ATM M and $\mathcal{K}$ a knowledge base consisting of the axioms (6)–(10) and (13)–(47). Then $c^0 \in \mathsf{C}_{\mathsf{acc}}(M)$ if and only if O is (finitely) satisfiable in $\mathcal{K}$.*

When analyzing the number of introduced axioms and their size, we see that their number is polynomial in n and their size is linear in n, where n is the size of the initial configuration. Hence, we get the following result.

Corollary 2. *The problem of (finite) concept satisfiability in the DL $\mathcal{SHI}^{\sqcap}$ is 2ExpTime-hard (and so are all the standard reasoning problems).*

The corresponding upper bound from [3] gives us the following result.

Corollary 3. *The problem of concept satisfiability in $\mathcal{SHI}^{\sqcap}$ and $\mathcal{SHIQ}^{\sqcap}$ is 2ExpTime-complete (and so are all the standard reasoning problems).*

Since the problem of KB satiafiability in $\mathcal{SHI}^{\sqcap}$ can be polynomially reduced to non-entailment of a union of conjunctive queries with at most two variables [5], we also get the following result.

Corollary 4. *The problem of entailment for unions of conjunctive queries in $\mathcal{SHI}$ is 2ExpTime-complete already for queries with at most two variables.*

5 $\mathcal{SHOIF}^{\sqcap}$ Is N2ExpTime-Hard

For proving the lower bound of reasoning in $\mathcal{SHOIF}^{\sqcap}$, we use a reduction from the double exponential domino tiling problem. We demonstrate how, by using $\mathcal{SHOIF}^{\sqcap}$ formulas, one can encode a $2^{2^n} \times 2^{2^n}$ grid-like structure illustrated in Figure 4. As in our tree-like structure in Figure 1 we will use four roles r_o^1, r_2^1, r_o^2, and r_e^2 with alternating directions to create the grid. Roles r_o^1 and r_e^1

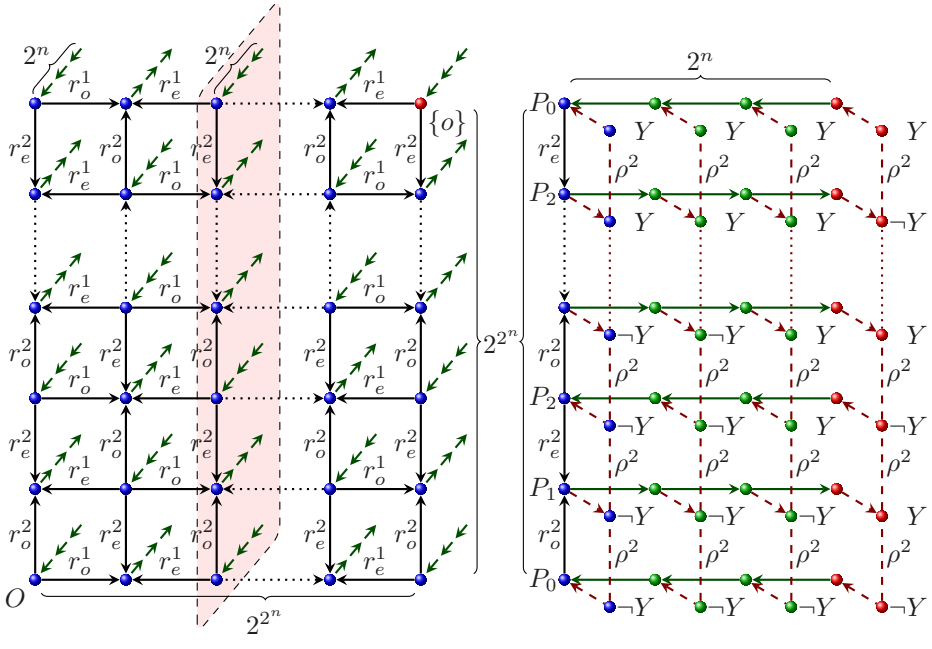


Fig. 4. A doubly exponential grid structure (left) and a detailed picture corresponding to the selected vertical slice in the grid (right)

induce horizontal edges and roles r_o^2 and r_e^2 induce vertical edges. The nodes of the grid are also partitioned on even and odd in a similar way as before: the odd nodes have only outgoing r-edges and the even nodes have only incoming r-edges. In fact our grid structure in Figure 4 is obtained from the tree structure in Figure 1 by merging the nodes that are reachable with the same number of horizontal and vertical edges up to a certain level; that is the nodes having the same "coordinates". The key idea of our construction is that in $\mathcal{SHOIF}^{\sqcap}$ it is possible to express doubly exponential counters for encoding the coordinates—a similar technique has been recently used in [2] for proving N2ExpTime-hardness of $\mathcal{SROIQ}$. We use a pair of counters to encode the coordinates of the grid: the counters are initialized in the origin O of the grid; the first counter is incremented across horizontal edges and the second counter is incremented across the vertical edges. We use nominals and inverse functional roles as in the hardness prove for $\mathcal{SHOIQ}$ [6] to enforce the uniqueness of the nodes with the same coordinates.

To store the values of the counters we will use exponentially long v-chains that grow from the nodes of the grid. The i^{th} element of the chain encodes the i^{th} bit of the horizontal counter using concept X and the i^{th} bit of the vertical counter using concept Y (see the right part of Figure 1). We will use auxiliary side h-chains like in our construction for $\mathcal{SHI}^{\sqcap}$ to connect the corresponding elements of the v-chains, which allows a proper incrementation of the counters.

In order to express the grid-like structure in Figure 4, we reuse all axioms (6)–(31) that define r-, v-, and h-chains, and add axioms to deal with the new counters and to merge the nodes with equal coordinates. First, we initialize both counters for the origin of our grid using auxiliary concepts Z^1 and Z^2:

$$O \sqsubseteq Z^1 \sqcap Z^2 \qquad Z^1 \sqsubseteq \neg X \sqcap \forall v.Z^1 \qquad Z^2 \sqsubseteq \neg Y \sqcap \forall v.Z^2 \tag{48}$$

Next, we introduce two concepts X^f and Y^f which express that the corresponding bit of the counter needs to be flipped in the successor value. Thus, the ending bit of the counter should always be flipped, while any other bit of the counter should be flipped if and only if the lower bit of the counter (accessible via v) is flipped from 1 to 0:

$$E \sqsubseteq X^f \sqcap Y^f \tag{49}$$

$$\exists v.(X \sqcap X^f) \sqsubseteq X^f \qquad \exists v.\neg(X \sqcap X^f) \sqsubseteq \neg X^f \tag{50}$$

$$\exists v.(Y \sqcap Y^f) \sqsubseteq Y^f \qquad \exists v.\neg(Y \sqcap Y^f) \sqsubseteq \neg Y^f \tag{51}$$

Additionally, we express that the values of X, Y, X^f, and Y^f agree across all elements of the same h-chain:

$$X \sqsubseteq \forall h.X \qquad \neg X \sqsubseteq \forall h.\neg X \qquad Y \sqsubseteq \forall h.Y \qquad \neg Y \sqsubseteq \forall h.\neg Y \tag{52}$$

$$X^f \sqsubseteq \forall h.X^f \qquad \neg X^f \sqsubseteq \forall h.\neg X^f \qquad Y^f \sqsubseteq \forall h.Y^f \qquad \neg Y^f \sqsubseteq \forall h.\neg Y^f \tag{53}$$

Finally, we express when the bits are flipped and when they are not flipped for the successor configurations using the property that the end elements of h-chains are related to exactly the corresponding elements of the successor chains

via the roles ρ^1 and ρ^2. The axioms are analogous to axioms (38) and (39) that propagate the information to the successor configurations:

$$Q_i \sqcap X^f \sqsubseteq (X \sqcap \forall\rho^1.[\neg Q_{[i+1]_3} \sqcup \neg X]) \sqcup (\neg X \sqcap \forall\rho^1.[\neg Q_{[i+1]_3} \sqcup X]) \tag{54}$$

$$Q_i \sqcap \neg X^f \sqsubseteq (X \sqcap \forall\rho^1.[\neg Q_{[i+1]_3} \sqcup X]) \sqcup (\neg X \sqcap \forall\rho^1.[\neg Q_{[i+1]_3} \sqcup \neg X]) \tag{55}$$

$$Q_i \sqcap Y^f \sqsubseteq (Y \sqcap \forall\rho^2.[\neg Q_{[i+1]_3} \sqcup \neg Y]) \sqcup (\neg Y \sqcap \forall\rho^2.[\neg Q_{[i+1]_3} \sqcup Y]) \tag{56}$$

$$Q_i \sqcap \neg Y^f \sqsubseteq (Y \sqcap \forall\rho^2.[\neg Q_{[i+1]_3} \sqcup Y]) \sqcup (\neg Y \sqcap \forall\rho^2.[\neg Q_{[i+1]_3} \sqcup \neg Y]) \tag{57}$$

The following formulas express that the counters are copied for other directions:

$$Q_i \sqsubseteq (X \sqcap \forall\rho^2.[\neg Q_{[i+1]_3} \sqcup X]) \sqcap (\neg X \sqcap \forall\rho^2.[\neg Q_{[i+1]_3} \sqcup \neg X]) \tag{58}$$

$$Q_i \sqsubseteq (Y \sqcap \forall\rho^1.[\neg Q_{[i+1]_3} \sqcup Y]) \sqcap (\neg Y \sqcap \forall\rho^1.[\neg Q_{[i+1]_3} \sqcup \neg Y]) \tag{59}$$

In order to determine whether the counters have reached the maximal value $2^{2^n} - 1$, we use concepts E^1 and E^2 that hold on the elements of v-chains if and only if X, respectively Y, hold for all v-successors until the end of the chain:

$$X \sqcap (E \sqcup \exists v.E^1) \sqsubseteq E^1 \qquad E^1 \sqsubseteq X \sqcap (E \sqcup \forall v.E^1) \tag{60}$$

$$Y \sqcap (E \sqcup \exists v.E^2) \sqsubseteq E^2 \qquad E^1 \sqsubseteq X \sqcap (E \sqcup \forall v.E^1) \tag{61}$$

In order to avoid creating r-successors after the maximal values of the counters are reached, we replace axioms (11) and (12) with (62) and (63):

$$Z \sqcap \mathrm{Odd} \sqsubseteq (E^1 \sqcup \exists r_e^1.\mathrm{Even}) \sqcap (E^2 \sqcup \exists r_e^2.\mathrm{Even}) \tag{62}$$

$$Z \sqcap \mathrm{Even} \sqsubseteq (E^1 \sqcup \exists r_o^{1^-}.\mathrm{Odd}) \sqcap (E^2 \sqcup \exists r_o^{2^-}.\mathrm{Odd}) \tag{63}$$

In order to merge the elements with the same coordinates, we first merge the elements that have the maximal values for both counters:

$$Z \sqcap E^1 \sqcap E^2 \sqsubseteq \{o\} \tag{64}$$

The preceding elements with the same coordinates are then merged by asserting functionality of the roles r^1 and r^2 that are respective superroles of r_e^1, $r_o^{1^-}$, r_e^2, and $r_o^{2^-}$ according to (13):

$$\mathsf{Func}(r^1) \qquad \mathsf{Func}(r^2) \tag{65}$$

Our complexity result for $\mathcal{SHOIF}^\sqcap$ is now obtained by a reduction from the bounded domino tiling problem. A *domino system* is a triple $D = (T, H, V)$, where $T = \{1, \ldots, k\}$ is a finite set of *tiles* and $H, V \subseteq T \times T$ are *horizontal* and *vertical matching relations*. A *tiling* of $m \times m$ for a domino system D with *initial condition* $c^0 = \langle t_1^0, \ldots, t_n^0 \rangle$, $t_i^0 \in T, 1 \leq i \leq n$, is a mapping $t\colon \{1, \ldots, m\} \times \{1, \ldots, m\} \to T$ such that $\langle t(i-1, j), t(i, j) \rangle \in H, 1 < i \leq m, 1 \leq j \leq m$, $\langle t(i, j-1), t(i, j) \rangle \in V, 1 \leq i \leq m, 1 < j \leq m$, and $t(i, 1) = t_i^0, 1 \leq i \leq n$. It is well known [13] that there exists a domino system D_0 that is N2ExpTime-complete for the following decision problem: given an initial condition c^0 of the size n, check if D_0 admits the tiling of $2^{2^n} \times 2^{2^n}$ for c^0.

In order to encode the domino problem on our grid, we use new atomic concepts $T_1, \dots, T_k$ for the tiles of the domino system D_0. The following axioms express that every element in our structure is assigned with a unique tile and that it is not possible to have horizontal and vertical successors that do not agree with the matching relations

$$\top \sqsubseteq T_1 \sqcup \dots \sqcup T_k \tag{66}$$

$$T_i \sqcap T_j \sqsubseteq \bot \qquad 1 \leq i < j \leq k \tag{67}$$

$$T_i \sqcap \exists r^1.T_j \sqsubseteq \bot \qquad \langle i, j\rangle \notin H \tag{68}$$

$$T_i \sqcap \exists r^2.T_j \sqsubseteq \bot \qquad \langle i, j\rangle \notin V \tag{69}$$

Finally, we express the initial condition of the grid:

$$O \sqsubseteq T_{t_1^0} \sqcap \forall r^1.(T_{t_2^0} \sqcap \forall r^1.(T_{t_3^0} \sqcap \forall r^1.(T_{t_4^0} \sqcap \dots \forall r^1.T_{t_n^0} \dots))) \tag{70}$$

Note that the size and the number of formulas that we have constructed is polynomial in the size of c^0. Since D_0 is fixed, we obtain a polynomial reduction from the doubly exponential domino tiling problem to the problem of $\mathcal{SHOIF}^\sqcap$ knowledge base satisfiability.

Theorem 3. *Let c^0 be an initial condition of size n for the domino system D_0 and $\mathcal{K}$ a knowledge base consisting of axioms (6)–(10), (13)–(31), and (48)–(70). Then D_0 admits the tiling of $2^{2^n} \times 2^{2^n}$ for c^0 if and only if O is (finitely) satisfiable in $\mathcal{K}$.*

Corollary 5. *The problem of (finite) concept satisfiability in the DL $\mathcal{SHOIF}^\sqcap$ is N2ExpTime-hard (and so are all the standard reasoning problems).*

6 Conclusions

Our investigation of the computational complexity of DLs with role conjunctions is motivated by the facts that (i) role constructors recently gained attention since the upcoming OWL2 standard supports a much richer set of role constructors and (ii) conjunctive query answering in a DL $\mathcal{L}$ is often reducible to the knowledge base satisfiability problem for $\mathcal{L}$ with role conjunctions (e.g., for $\mathcal{SHIQ}$ and $\mathcal{SHOQ}$ this is the case). We have shown that role conjunctions cause an exponential blowup for the DLs $\mathcal{SHI}^\sqcap$ and $\mathcal{SHOIF}^\sqcap$. The main culprit for this are inverse roles, which we show by proving ExpTime-completeness of $\mathcal{SHQ}^\sqcap$. Our results imply that conjunctive query entailment for $\mathcal{SHI}$ is 2ExpTime-hard already for a bounded number of variables in the query. The previously known proof for 2ExpTime-hardness [14] has unbounded number of variables in queries.

It remains an open question whether $\mathcal{SHOIF}^\sqcap$ is N2ExpTime-complete and so far even decidability is unknown. We think that the answer to this question can shed some light on the problem of decidability of conjunctive query entailment in $\mathcal{SHOIN}$ and, thus, OWL DL, which is a long-standing open problem.

References

1. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F. (eds.): The Description Logic Handbook. Cambridge University Press, Cambridge (2003)
2. Kazakov, Y.: $\mathcal{RIQ}$ and $\mathcal{SROIQ}$ are harder than $\mathcal{SHOIQ}$. In: Proc. of the 11th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR 2008). AAAI Press/The MIT Press (2008)
3. Glimm, B., Horrocks, I., Lutz, C., Sattler, U.: Conjunctive query answering for the description logic $\mathcal{SHIQ}$. J. of Artificial Intelligence Research 31, 151–198 (2008)
4. Lange, M., Lutz, C.: 2-ExpTime lower bounds for Propositional Dynamic Logics with intersection. J. of Symbolic Logic 70(5), 1072–1086 (2005)
5. Glimm, B., Kazakov, Y.: Role conjunctions in expressive description logics. Technical report, University of Oxford (2008), http://www.comlab.oxford.ac.uk//files/361/RoleConjunctions.pdf
6. Tobies, S.: Complexity Results and Practical Algorithms for Logics in Knowledge Representation. PhD thesis, RWTH Aachen (2001)
7. Schild, K.: A correspondence theory for terminological logics: preliminary report. In: Proc. of the 12th Int. Joint Conf. on Artificial Intelligence (IJCAI 1991) (1991)
8. Kazakov, Y., Motik, B.: A resolution-based decision procedure for $\mathcal{SHOIQ}$. J. of Automated Reasoning 40(2–3), 89–116 (2008)
9. Hustadt, U., Motik, B., Sattler, U.: Reducing $\mathcal{SHIQ}^-$ Description Logic to Disjunctive Datalog Programs. In: Proc. of the 9th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR 2004) (2004)
10. Sipser, M.: Introduction to the Theory of Computation, 2 edn. Course Technology (February 2005)
11. Chandra, A.K., Kozen, D.C., Stockmeyer, L.J.: Alternation. J. of the ACM 28(1), 114–133 (1981)
12. Tobies, S.: The complexity of reasoning with cardinality restrictions and nominals in expressive description logics. J. of Artificial Intelligence Research 12 (2000)
13. Börger, E., Grädel, E., Gurevich, Y.: The classical decision problem. J. of Logic, Language and Information 8(4), 478–481 (1999)
14. Lutz, C.: Inverse roles make conjunctive queries hard. In: Proc. of the 2007 Description Logic Workshop (DL 2007) (2007)

Default Logics with Preference Order: Principles and Characterisations

Tore Langholm

Royal Norwegian Naval Academy, Box 83 Haakonsvern, N-5886 Bergen
tore.langholm@sksk.mil.no

Abstract. Practical use of default logics requires mechanisms to select the more suitable extensions from out of the several often allowed by a classical default theory. An obvious solution is to order defaults in a preference hierarchy, and use this ordering to select preferred extensions. The literature contains many suggestions on how to implement such a scheme. The problem is that they yield different results: all authors agree that preferred extensions employ preferred defaults, but this apparent agreement hides differences in lower level decisions. While motivations for these are rarely discussed, their consequences for overall behaviour are wide-ranging. This paper points towards standardisation, discussing principles that ought to hold and then working top-down to determine lower level details. We present characterisations, uncover anomalies of existing approaches, and suggest repairs.

We build on work by Brewka and Eiter [4], who first identified some of the desiderata discussed here. A slightly modified version of the notion of preferred extension proposed by these authors, and one by Delgrande and Schaub [5], are identified as the most and least inclusive notions of extension satisfying all desiderata. We point out that these two (in the literature previously termed "descriptive" and "prescriptive", respectively) differ along two rather independent dimensions, and two additional notions are then identified, representing the remaining possibilities.

1 Introduction

A classical default theory in the tradition of Reiter [11] typically allows several extensions, of which one or a few will be deemed more appropriate in concrete situations. Classical default logic does not provide sufficient means for carrying out such comparison between extensions within the theory itself. One strategy for accommodating appropriate comparison criteria within the theory, has been the introduction of preference orders on the set of defaults [1,2,4,5,10,12]. Very roughly, the extensions obtained using more preferred defaults are then to be viewed as the better ones, i.e., the ones to be assigned an OK stamp. When it comes to the precise mechanisms by which an ordering of defaults is translated

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 406–420, 2008.

into an acceptability criterion on extensions, there has, however, been remarkably little agreement. In the terminology we shall use, there has been a wide variation among the *extension predicates* proposed by different authors: an extension predicate (EP for short) is something that determines, for any ordered default theory, the set of acceptable extensions.

A landmark contribution was the article by Brewka and Eiter [4], which identifies some salient properties to look for in reasonable EPs. The majority of approaches found in the literature at the time did not pass their test, but a couple did, and indeed the range of possible EPs one can define that do pass, remains wide and encompassing. Present work discusses additional, and in our view equally reasonable, principles to further narrow down this space of possible EPs, and then shows how to use these principles to characterise concrete EPs. What we hope to see as the eventual outcome of such work, is a typology of sort, identifying possible ways of interpreting the preference order, and determining in each case the one and only appropriate EP to use. We believe that many of the central issues are still to be settled, one clear indication of this being the fact that two prominent EPs in the literature fail to comply with a principle of *Base Logic Invariance* introduced here, an extremely weak requirement saying that intuitively vacuous defaults of a type that merely restate logical inferences which in any case are taken care of by the underlying monotonic logic, should have no bearing on the set of acceptable extensions.

Original contributions of the present work are threefold. First and foremost, there is a methodological twist; rather than passing from one EP to the next, each time detecting unforeseen defects in previous approaches and seeking to alleviate these by smart fixes, we propose a less ad-hoc, more top-down procedure, starting from general principles and looking to identify the right EP satisfying these. Secondly we discuss a collection of such principles that one should want to see in the treatment of defaults. This second contribution is not complete; the principles described do not suffice to identify any one unique EP, but it does take us quite a bit of the way. Our third contribution, then, is a study of the strongest and weakest EPs satisfying various collections of principles. This last part includes characterisations of such abstractly defined EPs, stated in very concrete, algorithmic terms.

2 Basic Notions

For present discussions there is no need to fix the underlying ("base") monotonic logic to be any of the typical candidates usually considered, such as propositional or predicate logic of some flavour. We merely assume the existence of some set $\mathcal{L}$ of formulas together with an inference relation of the format $E \vdash \varphi$, with subsets and members of $\mathcal{L}$ to the left and right, respectively. For any $E \subseteq \mathcal{L}$, we write $Th(E)$ for the set $\{\varphi \mid E \vdash \varphi\}$, and, following Tarski [14], assume that $\vdash$ behaves such that Th is a closure operator.[1] We also require formulas $\top, \bot$ in $\mathcal{L}$, such that $\top \in Th(\emptyset)$ and $Th(\{\bot\}) = \mathcal{L}$. As usual, equivalence means mutual

[1] $E \subseteq Th(E)$ and $Th(Th(E)) = Th(E)$ and $Th(E) \subseteq Th(F)$ for all $E \subseteq F \subseteq \mathcal{L}$.

inferrability, hence φ, ψ are equivalent iff $\varphi \vdash \psi$ and $\psi \vdash \varphi$. It is convenient to assume that each formula is equivalent to infinitely many others.[2]

A set E of formulas is **consistent** if $E \nvdash \bot$, i.e., iff $Th(E) \neq \mathcal{L}$, and **inconsistent** if it is not consistent. Furthermore, φ is **consistent** (**inconsistent**) **with** E, written $E \Diamond \varphi$ ($E \not\Diamond \varphi$), if $E \cup \{\varphi\}$ is consistent (inconsistent). We write $E \vdash F$ if $F \subseteq Th(E)$, i.e., if $E \vdash \varphi$ for every $\varphi \in F$.

A **default rule** (or just **default**) is of the form $\alpha : \beta/\gamma$, where $\alpha, \beta, \gamma \in \mathcal{L}$. The components α, β and γ are the **prerequisite**, **justification** and **consequence**, respectively. The letter δ (and δ', δ_0, etc.) shall be used to range over defaults. Frequently we shall want to refer to the individual components of a given δ; rather than write $pre(\delta)$, $just(\delta)$, $cons(\delta)$ or the like, we shall then, for better readability, adopt a convention linking the four variables $\alpha, \beta, \gamma, \delta$ in such a way that δ is at any time understood to be the default $\alpha : \beta/\gamma$. Similarly for $\alpha', \beta', \gamma', \delta'$ and $\alpha_0, \beta_0, \gamma_0, \delta_0$, etc. Defaults of the form $\alpha : \top/\gamma$ are **justification free**; for these we use the simpler notation α/γ.

If D is a set of defaults and E is a set of formulas, then $GD(D, E)$ is defined to be the set of defaults δ in D such that $E \vdash \alpha$ and $E \Diamond \beta$. Now the semantics of defaults is, at least partly, captured by a notion of D-closure: for any set D of defaults, a set E of formulas is **D-closed** if $E \vdash \gamma$ for every $\delta \in GD(D, E)$.

A **default theory** is a pair (D, W) consisting of a set W of formulas and a set D of defaults. An extension of the theory is any D-closed formula set E containing W and satisfying the closure condition $Th(E) = E$ and which is, in an intuitive sense, reachable from W using D in a careful way that ensures each step to remain justifiable at later stages.[3] To simplify discussions we assume that D is always finite;[4] extensions can then be defined through a notion of proof sequence, defined next. For any set W of formulas and set A or sequence u of defaults, let W_A and W_u be the sets $W \cup \{\gamma \mid \delta \in A\}$ and $W \cup \{\gamma \mid \delta \in u\}$. Now, the sequence u of defaults is a **proof sequence** (PS) if it contains no repetitions and W_u is D-closed, and for any $s\delta \preceq u$ we have $W_s \vdash \alpha$ and $W_u \Diamond \beta$. Here, $v \preceq u$ expresses that v is an initial segment of u, i.e., that $u = vw$ for some (possibly empty) sequence w. It is well known that the following definition is equivalent to Reiter's original when applied to finite Ds.

[2] This condition is met by any logic containing a binary connective $\odot$ (e.g., $\wedge$ or $\vee$) such that φ and $(\varphi \odot \varphi)$ and $((\varphi \odot \varphi) \odot \varphi)$ etc., are all distinct and equivalent. There are several reasons for wanting this assumption, one is that some authors take an "intensional" view of defaults, allowing two defaults to share the same prerequisite, justification and consequence and yet be distinct, hence allowing for the theoretical possibility of having "the same" default appearing at different places in the preference hierarchy. Under the present assumption, this is easily simulated using distinct copies of the formulas involved.

[3] Following Reiter [11], the extensions of (D, W) are the fixed-points of an operator mapping any set S of formulas to the smallest $E \subseteq \mathcal{L}$ containing W, such that $E = Th(E)$ and such that for each $\delta \in D$, if $E \vdash \alpha$ and $S \Diamond \beta$, then $E \vdash \gamma$.

[4] The relevant distinctions to be discussed already appear in this restricted setting, hence the technicalities introduced by infinite D's would, in our view, only serve as irrelevant distractions away from the central points to be conveyed.

Definition 1. E is an **extension** of (D, W) if $E = Th(W_u)$ for a PS u.

In this case we write $(D, W) \not\mapsto E$. All defaults occurring in u will obviously be members of $GD(D, W_u)$, but u does not necessarily contain *all* of $GD(D, W_u)$. Remaining members of $GD(D, W_u)$ would, on the other hand, contribute nothing additional, as W_u is still required to be D-closed. Proof sequences that do contain all of $GD(D, W_u)$ are **full**; also the following is known and easily checked.

Lemma 1. $(D, W) \not\mapsto E$ *iff* $E = Th(W_u)$ *for some full PS* u.

An **ordered default theory** is a triple $(D, W, <)$ where (D, W) is a classical default theory and $<$ is a strict, partial order on D. We interpret $<$ as a preference order[5] on defaults, in such a way that δ_1 is preferred to δ_2 if $\delta_1 < \delta_2$. Suppose $D_0 \subseteq D$, then δ_0 is a **minimal element of** (or **minimal in**) D_0 if $\delta_0 \in D_0$, and $\delta_1 < \delta_0$ for no $\delta_1 \in D_0$. Note that any non-empty, finite $D_0 \subseteq D$ has at least one minimal element.

We shall sometimes refer to the subtheory of $(D, W, <)$ containing only the defaults occurring in some given subset D_0 of D, and write $(D_0, W, <)$ for this, although $<$ may in fact not be a subset of $D_0 \times D_0$. What we shall mean then, is of course $(D_0, W, <_0)$, where $<_0$ is the restriction of $<$ to D_0.

Let $(D, W, <)$ be any ordered theory and let A be a subset of D, we then define $\uparrow A$ to be the set $\{\delta' \in D \mid \delta \in A,\ (\delta = \delta' \text{ or } \delta < \delta')\}$. A subset $D_0 \subseteq D$ is **preferentially closed** if $\delta \in D_0$ whenever $\delta < \delta_0$ and $\delta_0 \in D_0$.

An **extension predicate (EP)** determines, for any ordered default theory, an associated set of acceptable extensions. We shall use the symbols $\not\mapsto_x$ and $\not\mapsto_y$ for arbitrary EPs, and write $(D, W, <) \not\mapsto_x E$ if E is an acceptable extension of $(D, W, <)$ according to $\not\mapsto_x$. We say that $\not\mapsto_y$ **contains** $\not\mapsto_x$, or $\not\mapsto_y$ is **at least as weak as** $\not\mapsto_x$, or $\not\mapsto_x$ is **at least as strong as** $\not\mapsto_y$, if $(D, W, <) \not\mapsto_x E$ always implies $(D, W, <) \not\mapsto_y E$.

3 Core Principles

This paper primarily discusses principles of *Order Compliance* – in which way or ways the extensions of an ordered default theory should comply with the ordering on defaults. To carry out such discussions with sufficient rigour, we need a set of ground rules, stipulating that candidate EPs do not deviate from classical default logic in arbitrary ways. For this purpose we introduce the following **core principles**, collectively referred to as "CP".

Normality If $(D, W, <) \not\mapsto_x E$ then $(D, W) \not\mapsto E$
Null-Preference If $(D, W) \not\mapsto E$ then $(D, W, \emptyset) \not\mapsto_x E$
Relevance If $(D \setminus \{\delta\}, W, <) \not\mapsto_x E$ and $(E \not\vdash \alpha$ or $W \not\vdash \beta)$ then $(D, W, <) \not\mapsto_x E$.
Positivity If $(D, W, <) \not\mapsto_x E$ and $(E \not\vdash \alpha$ or $E \not\vdash \beta)$ then $(D \setminus \{\delta\}, W, <) \not\mapsto_x E$.
Interpolation If $(D, W, <) \not\mapsto_x E$ and $E \vdash V$ and $V \vdash W$ then $(D, V, <) \not\mapsto_x E$.

[5] Beware that some authors adopt the opposite convention, according to which δ_2 is more preferred if $\delta_1 < \delta_2$.

A common characteristic of the five is that they hardly at all refer to the preference order; they mostly record uniformities seen in classical default logic, and stipulate that these carry over to the ordered version. The list is partly compiled from the literature;[6] we hope it will be clear from the following that they express weak, uncontroversial conditions that would, in less formal settings, be dismissed as trivial. Note in particular that it is not a list of pragmatically useful properties ensuring easy processing or convenient numbers of accepted extensions – these are all minimal requirements without which a candidate logic would in our view lack sufficient semantic transparency.

Normality says that the extensions of any ordered theory are also extensions of the corresponding classical, unordered theory: introducing priorities should only have the effect of narrowing down the set of possible extensions – never the opposite one of introducing new. In the presence of Normality, Null-Preference goes on to say that an ordered theory with no preferences recorded should behave exactly as the corresponding classical theory: ordered default logic employs no other device than priorities for narrowing down the set of extensions.[7]

Relevance expresses the intuition that any extension should remain so after the introduction of new defaults dealing with different circumstances, i.e., containing prerequisites that don't show up in the present case, or justifications contradicted at the outset. It really consists of two principles, of which the first is nothing but Principle II introduced by Brewka and Eiter.[8] For an example, consider a default theory used for medical diagnosis, and suppose the available facts and defaults yield an extension which in no way indicates that kidneys are encountering problems, such as overloading due to internal or external factors. Now by the first half of Relevance this extension remains valid after the introduction of new defaults applying only in cases where kidney overloading is indicated. A slight modification will do for illustration of the second half: suppose the available facts and defaults yield a particular extension, and that available facts show the kidneys to be in perfect working order. By Relevance this extension remains valid after the introduction of new defaults applying only in cases where kidney failure cannot be excluded.

Positivity is a similar pair, expressing that a default can only contribute positively to something being an extension: if a theory yields a particular extension and a given default available in the theory clearly does not contribute to this, either because its prerequisite is not implied by the extension or because its justification is contradicted by it, then the extension should remain so when the default is removed from the theory. Simple adaptations of the above examples could serve to illustrate this; we leave the task to the interested reader.

Interpolation says that if available facts and defaults yield a particular extension, and then new, independent facts (which go in W) are obtained which are

[6] Engan [6] discusses the first two under the names of *Classical Subset* and *Empty Order,* respectively.

[7] This is not to say that other approaches for resolving multiple extensions are not to be considered – only that such approaches should be recognized as that, i.e., other approaches. Present work considers what can be achieved with priorities alone.

[8] The principle is exactly the same, although the wording is different.

already implied by the extension, then the extension should remain an extension of the modified theory. Returning to kidneys: if an extension is obtained, yielding a diagnosis which, as part of the picture, concludes that liver and kidneys are overloaded, then this diagnosis remains valid if someone turns up with independent evidence of kidney overloading – provided this evidence does not contribute additional details not already present in the extension.

4 Minimal and Weak Order Compliance

A more crucial discussion concerns conditions that relate EPs to the ordering, The following, introduced by Brewka and Eiter [4], is of the sort:

Principle I. If E_1 and E_2 are distinct, classical extensions of (D, W), where $GD(D, E_1)$ and $GD(D, E_2)$ are $R \cup \{\delta_1\}$ and $R \cup \{\delta_2\}$ respectively, for $\delta_1 < \delta_2$, then E_2 is not an extension of the ordered theory $(D, W, <)$.

If the precedence order is to have any role to play at all in the selection of acceptable extensions, then this would seem an absolutely minimal requirement. The condition itself does, however, have the flavour of a useful but restricted test criterion (note the very limited cases to which it applies) rather than of a well-rounded principle, and we shall discuss instead some more general conditions that, together with CP, will ensure Principle I. Consider first:

Minimal Order Compliance. Suppose $(D, W, <) \not\rightarrow_x E$, and suppose moreover that $E \vdash \alpha$ and $E \not\vdash \gamma$ for some $\delta \in D$. Then $W_{D_0} \not\sim \beta$ for some $D_0 \subseteq D \setminus \uparrow\{\delta\}$ such that $E \vdash \alpha_0$ and $E \Diamond \beta_0$ for every $\delta_0 \in D_0$.

This is perhaps best considered in conjunction with Normality: if E is allowed as an extension of $(D, W, <)$, and a given default $\delta \in D$ does not fire although its precondition α is satisfied, then by Normality we know that E refutes β. Minimal Order Compliance makes, however, the stronger claim that in this case a subset of E may also be identified, which refutes β, and contains just W and consequences of defaults in D of which none are less preferred than δ. This suffices to ensure Principle I:

Proposition 1. *Any EP satisfying CP and Minimal Order Compliance also satisfies Principle I.*

Due to space restrictions, the proof is omitted here. It can be found, together with proofs of all results listed below, in Langholm [9].

Minimal Order Compliance is closely related to the EP proposed by Brewka and Eiter [4]; their Proposition 15 can be rephrased to say that, *in the special case of totally ordered default theories,* their EP is the weakest possible (the one allowing most[9] extensions) that satisfies Minimal Order Compliance and Normality. For our purposes it suffices to use this as the definition:

[9] Allowing all extensions allowed by any other EPs satisfying these conditions.

Definition 2. $(D, W, <) \nvdash_{BE} E$ iff $(D, W) \nvdash E$ and for every $\delta \in D$, if $E \vdash \alpha$ and $E \nvdash \gamma$ then $W_{D_0} \not\!\!\downarrow \beta$ for some $D_0 \subseteq D \setminus \uparrow\{\delta\}$ such that $E \vdash \alpha_0$ and $E \Diamond \beta_0$ for every $\delta_0 \in D_0$.

Note carefully that $\nvdash_{BE}$ is *not* the EP introduced by Brewka and Eiter, but coincides on totally ordered theories. It can be shown to satisfy CP. Since by definition it is the weakest EP satisfying Normality and Minimal Order Compliance, it is also the weakest satisfying CP and Minimal Order Compliance.

We find two defects with $\nvdash_{BE}$. One of these it shares with Brewka and Eiter's EP, the other it doesn't. We consider the last first. To avoid potentially confusing overloading of the word "extension," we write "linearisation of $<$" to mean a total (linear) extension of $<$.

Definition 3. The EP $\nvdash_x$ is **sharp** if for any ordered theory $(D, W, <)$ and set E of formulas, $(D, W, <) \nvdash_x E$ iff $(D, W, <') \nvdash_x E$ for a linearisation $<'$ of $<$.

The following example shows that $\nvdash_{BE}$ is not sharp.

Example 1. Consider $(D, \emptyset, <)$, with D containing the defaults

$$\begin{array}{ll} \delta_1 = \top : p/p & \delta_1' = \top : q/q \\ \delta_2 = \top : \neg q/\neg q & \delta_2' = \top : \neg p/\neg p \end{array}$$

where $\delta_1 < \delta_2$ and $\delta_1' < \delta_2'$. It is easy to see that either $\delta_1 <' \delta_2'$ or $\delta_1' <' \delta_2$ for any linearisation $<'$ of $<$, hence any $\nvdash_x$ satisfying Minimal Order Compliance will be such that $E \vdash p$ or $E \vdash q$ whenever $(D, \emptyset, <') \nvdash_x E$. However, as Minimal Order Compliance only "talks about one δ_1 at a time," it fails to capture the relevant conditions when faced with partially ordered theories. In particular, $(D, \emptyset, <) \nvdash_{BE} Th(\{\neg p, \neg q\})$, although $(D, \emptyset, <') \nvdash_{BE} Th(\{\neg p, \neg q\})$ for no linearisation $<'$ of $<$.

Brewka and Eiter in effect[10] define their notion of preferred extension to be the unique sharp EP that coincides with $\nvdash_{BE}$ on totally ordered theories. The defaults in the next example are totally ordered; hence what is shown there about $\nvdash_{BE}$ applies equally for Brewka and Eiter's notion.

Example 2. Let D contain

$$\begin{array}{l} \delta_0 = p/p \\ \delta_1 = \top : \neg p/\neg p \\ \delta_2 = \top : p/p \end{array}$$

where $\delta_0 < \delta_1 < \delta_2$. Then $(D, \emptyset)$ has the extensions $Th(\{p\})$ and $Th(\{\neg p\})$, of which *both* are extensions of $(D, \emptyset, <)$ in the sense of $\nvdash_{BE}$, i.e., such that $(D, \emptyset, <) \nvdash_{BE} Th(\{p\})$ as well as $(D, \emptyset, <) \nvdash_{BE} Th(\{\neg p\})$.

The example illustrates the behaviour of $\nvdash_{BE}$ that vacuous defaults such as φ/φ may affect which extensions are accepted. Hence $\nvdash_{BE}$, as well as Brewka

[10] Brewka and Eiter [4], Definition 3 and Proposition 15.

and Eiter's EP, violate the "only if" parts (but satisfy the "if" parts) of both principles below.[11]

Trivial Invariance. If δ is any justification free default of the form φ/φ, then $(D, W, <) \nvdash_x E$ iff $(D \setminus \{\delta\}, W, <) \nvdash_x E$.

Base Logic Invariance. If δ is any justification free default of the form φ/ψ where $\varphi \vdash \psi$, then $(D, W, <) \nvdash_x E$ iff $(D \setminus \{\delta\}, W, <) \nvdash_x E$.

Horty [8] recently introduced an EP for normal default theories which is somewhat related to Brewka and Eiter's, and which violates a version of trivial invariance for normal default theories in essentially the same way.[12] Later we shall note that the EP introduced by Delgrande and Schaub [5] has the opposite behaviour; it satisfies the "only if" parts but violates the "if" parts of these principles.

Now consider the following strengthening of Minimal Order Compliance.

> Suppose $(D, W, <) \nvdash_x E$, and suppose moreover that $E \vdash \alpha$ and $E \nvdash \gamma$ for some $\delta \in D$. Then $(D_0, W, <) \nvdash_x E_0$ and $E_0 \not\vdash \beta$ for some $E_0 \subseteq E$ and $D_0 \subseteq D \setminus \uparrow\{\delta\}$.

This version clearly implies the previous. What is different here, is that the subset of E refuting β must be formed from W and the consequences of not just any set of sufficiently preferred defaults in $GD(D, E)$, but rather of such a set of defaults that are applicable to the case, i.e., which jointly fire from W.

This modification was motivated by Base Logic Invariance. As it stands here, however, the condition still only talks about one default at a time. The appropriate generalisation fixing this, is the following.

Weak Order Compliance. Suppose $(D, W, <) \nvdash_x E$, and suppose $N \neq \emptyset$ is a set of defaults in D such that $E \vdash \alpha$ and $E \nvdash \gamma$ for each $\delta \in N$. Then $(D_0, W, <) \nvdash_x E_0$ and $E_0 \not\vdash \beta$ for some $\delta \in N$, $E_0 \subseteq E$ and $D_0 \subseteq D \setminus \uparrow N$.

It can be shown that this condition is equivalent to the previous, when applied to sharp EPs satisfying CP. To sum up; the step from Minimal to Weak Order Compliance was motivated by the two defects of $\nvdash_{BE}$ seen in examples 1–2.

Definition 4. Let $(D, W, <)$ be an ordered theory; the PS u is **weakly obedient** if for any δ_0, δ_1 in D,

$$W_u \vdash \gamma_0 \text{ if } \delta_0 < \delta_1 \text{ and } r\delta_1 \preceq u \text{ and } W_u \vdash \alpha_0 \text{ and } W_r \Diamond \beta_0.$$

E is a **weakly obedient extension** of $(D, W, <)$, written $(D, W, <) \nvdash_W E$, if $E = Th(W_u)$ for some weakly obedient PS u.

[11] It violates the first, which is a special case of the second. In fact, it is easily seen that any classical extension can be made acceptable in the sense of $\nvdash_{BE}$ by the introduction of additional, sufficiently preferred defaults of the form φ/φ.

[12] The normal version of a trivial default would be $\varphi : \varphi/\varphi$. In a modified version of Example 2 in which $\delta_0 = p : p/p$ (and with the opposite ordering, cf. footnote no. 5) $\{\delta_0, \delta_2\}$ would count as a "proper scenario," yielding the extension $Th(\{p\})$.

Informally, a PS is weakly obedient if at any step it "weakly obeys" all defaults preferred to the one actually being applied, in the sense that if the justification of such a preferred default is consistent with the knowledge acquired so far, and if its prerequisite turns out to be derivable from the knowledge eventually assembled, then its conclusion should also follow from that eventual knowledge.

The reader is invited to check that $\mathrel{\sim\!\!\!\!\!\rightarrow}_W$ coincides with the authors' own proposed EP on all relevant examples[13] listed by Brewka and Eiter [4]. It also satisfies all principles discussed so far. Moreover, the principles can be used to characterise $\mathrel{\sim\!\!\!\!\!\rightarrow}_W$; it is at least as weak as (contains) any other EP satisfying Weak Order Compliance and CP:

Proposition 2. *$\mathrel{\sim\!\!\!\!\!\rightarrow}_W$ is sharp and satisfies Base Logic Invariance, CP and Weak Order Compliance.*

Proposition 3. *$\mathrel{\sim\!\!\!\!\!\rightarrow}_W$ is the weakest EP satisfying CP and Weak Order Compliance.*

Again, we refer to Langholm [9] for proofs.

5 Interlude: Blind Rejection and Empty Inclusion

Note that Weak Order Compliance and CP do not select $\mathrel{\sim\!\!\!\!\!\rightarrow}_W$ as the only possible EP, they only define a range with $\mathrel{\sim\!\!\!\!\!\rightarrow}_W$ located at one end as the most permissive candidate. To narrow the range further down, one should look for other principles. This should be done with utmost caution – it is not difficult to find a priori plausible principles that harbour unforeseen consequences. The following two principles are cases in point; each is quite appealing when considered in isolation, but it takes almost nothing to contradict one given the other. The pair in fact represents a crossroad for further directions.

Blind Rejection. If $(D, W, <) \mathrel{\sim\!\!\!\!\!\rightarrow}_x E$ and $E \not\vdash \beta$, then also $(D', W, <') \mathrel{\sim\!\!\!\!\!\rightarrow}_x E$, where $(D', W, <')$ is identical to $(D, W, <)$, except that δ has been replaced in the preference hierarchy by a default δ' with the same prerequisite and justification as δ, but possibly with a different consequence.[14]

Empty Inclusion. If $(D \setminus \{\delta\}, W, <) \mathrel{\sim\!\!\!\!\!\rightarrow}_x E$ and $E \vdash \gamma$, then $(D, W, <) \mathrel{\sim\!\!\!\!\!\rightarrow}_x E$.

[13] Brewka and Eiter [4] consider the numbered examples 4, 5, 8, 13, 14, 17 and 22, the intermittent numbers being reserved for definitions and propositions. Of these, Example 22 uses an extended format for which current definitions do not apply. In the remaining 6 examples, which are used to motivate their approach, and which all contain totally ordered theories, $\mathrel{\sim\!\!\!\!\!\rightarrow}_W$ (and in fact also $\mathrel{\sim\!\!\!\!\!\rightarrow}_O$ introduced below) and $\mathrel{\sim\!\!\!\!\!\rightarrow}_{BE}$, and hence also Brewka and Eiter's own EP, coincide.

[14] A more formal phrasing goes as follows: let $(D, <)$ and $(D', <')$ be two partially ordered sets of defaults for which there is an order-isomorphism h from the first onto the second, and suppose there is a $\delta \in D$ such that $h(\delta_0) = \delta_0$ for every $\delta_0 \in D \setminus \{\delta\}$ and for which $h(\delta) = \delta'$, where $\alpha = \alpha'$ and $\beta = \beta'$. Now if $(D, W, <) \mathrel{\sim\!\!\!\!\!\rightarrow}_x E$ and $E \not\vdash \beta$, then also $(D', W, <') \mathrel{\sim\!\!\!\!\!\rightarrow}_x E$.

The intuition behind Blind Rejection is somewhat procedural in spirit, referring to some process walking through the preference hierarchy and selecting defaults based on their prerequisites and justifications, while – prior to each selection – keeping a blind eye to what consequence might hide at the end of the default: in particular, if it was OK to reject any particular default, then the corresponding move would also be OK in the almost identical circumstances which differ only in what was in fact contained in the consequence of the rejected default.

The intuition behind Empty Inclusion is perhaps more purely declarative: if an extension is acceptable, it should remain so when new defaults are thrown in, containing consequences already implied by this extension.

It can be seen that the corresponding versions for unordered theories are both valid principles in classical default logic, but in their full versions they clash:

Proposition 4. *No EP satisfies all of the four principles Blind Rejection, Empty Inclusion, Null-Preference and Minimal Order Compliance.*

Proof. Consider the defaults

$$\delta_0 = \top : \neg p/q \qquad \delta_1 = \top/p$$
$$\delta_0' = \top : \neg p/\neg p \qquad \delta_2 = \top/q$$

Then $(\{\delta_1, \delta_2\}, \emptyset, \emptyset) \not\hookrightarrow_x Th(\{p, q\})$ if $\not\hookrightarrow_x$ satisfies Null-Preference. Hence also $(\{\delta_0, \delta_1, \delta_2\}, \emptyset, <) \not\hookrightarrow_x Th(\{p, q\})$ if $\not\hookrightarrow_x$ satisfies Empty Inclusion as well, where $<$ prefers δ_0 to δ_1 and δ_2, but keeps the others unordered. Further assuming Blind Rejection, we obtain $(\{\delta_0', \delta_1, \delta_2\}, \emptyset, <') \not\hookrightarrow_x Th(\{p, q\})$, where $<'$ prefers δ_0' to δ_1 and δ_2, in direct violation of minimal, as well as weak, Order Compliance. □

The next result follows directly from the definitions.

Proposition 5. $\not\hookrightarrow_W$ *satisfies Empty Inclusion.*

Corollary 1. $\not\hookrightarrow_W$ *is the weakest EP satisfying CP, Weak Order Compliance and Empty Inclusion.*

6 Order Compliance and Closed Order Compliance

In this section we consider an EP $\not\hookrightarrow_O$ that is slightly stronger than $\not\hookrightarrow_W$, and which satisfies Blind Rejection rather than Empty Inclusion. Comparing corollaries 1 (above) and 2 (below), one could say that $\not\hookrightarrow_O$ is the "blind rejective" version of $\not\hookrightarrow_W$. We also consider alternative characterisations, using the following stronger principles of Order Compliance:

Order Compliance. Suppose $(D, W, <) \not\hookrightarrow_x E$ and suppose $N \neq \emptyset$ is a set of defaults in D such that $E \vdash \alpha$ and $E \not\vdash \beta$ for each $\delta \in N$. Then $(D_0, W, <) \not\hookrightarrow_x E_0$ and $E_0 \not\vdash \beta$ for some $\delta \in N$, $E_0 \subseteq E$ and $D_0 \subseteq D \setminus \uparrow N$.

Closed Order Compliance. Suppose $(D, W, <) \not\hookrightarrow_x E$ and suppose $N \neq \emptyset$ is a set of defaults in D such that $E \vdash \alpha$ and $E \not\vdash \beta$ for each $\delta \in N$. Then $(D_0, W, <) \not\hookrightarrow_x E_0$ and $E_0 \not\vdash \beta$ for some $\delta \in N$, $E_0 \subseteq E$ and preferentially closed $D_0 \subseteq D \setminus N$.

Order Compliance differs from Weak Order Compliance by replacing $E \nvdash \gamma$ by the (under the circumstances) weaker condition $E \nvdash \beta$ in the antecedent, hence it applies in more cases. Closed Order Compliance is the further strengthening in which the D_0 in the consequent is required to be preferentially closed. We first note (proof omitted here) that Order Compliance and Weak Order Compliance are the same condition when applied to EPs satisfying Blind Rejection and CP.

Proposition 6. *Any EP satisfying Weak Order Compliance, Blind Rejection and CP, also satisfies Order Compliance.*

Definition 5. Let $(D, W, <)$ be an ordered theory; the PS u is **obedient** if for any δ_0, δ_1 in D,

$$\delta_0 \text{ is in } u \text{ if } \delta_0 < \delta_1 \text{ and } r\delta_1 \preceq u \text{ and } W_u \vdash \alpha_0 \text{ and } W_r \Diamond \beta_0.$$

E is an **obedient extension** of $(D, W, <)$, written $(D, W, <) \mathrel{\sim\!\!\!\!\!\rightarrow}_O E$, if $E = Th(W_u)$ for some obedient PS u.

Informally, a PS is obedient if at any step it "obeys" all defaults preferred to the one actually being applied, in the sense that if the justification of such a preferred default is consistent with the knowledge acquired so far, and if its prerequisite turns out to be derivable from the knowledge eventually assembled, then the preferred default should be included in the PS. It is worth noting that the following variant of the condition in Definition 5 is in fact equivalent.

Lemma 2. $(D, W, <) \mathrel{\sim\!\!\!\!\!\rightarrow}_O E$ *iff* $E = Th(W_u)$ *for some full and obedient PS* u.

The following example illustrates the difference between $\mathrel{\sim\!\!\!\!\!\rightarrow}_O$ and $\mathrel{\sim\!\!\!\!\!\rightarrow}_W$.

Example 3. Let D contain

$$\delta_0 = \top : \neg p/q$$
$$\delta_1 = \top / q$$
$$\delta_2 = \top / p$$

where $\delta_0 < \delta_1 < \delta_2$. Then the PS $\delta_1\delta_2$ is weakly obedient but not obedient. Hence $(D, \emptyset, <) \mathrel{\sim\!\!\!\!\!\rightarrow}_W Th(\{p, q\})$ but not $(D, \emptyset, <) \mathrel{\sim\!\!\!\!\!\rightarrow}_O Th(\{p, q\})$.

Brewka and Eiter [4] consider no example of this sort, and again the reader is encouraged to check that $\mathrel{\sim\!\!\!\!\!\rightarrow}_O$, like its close relative $\mathrel{\sim\!\!\!\!\!\rightarrow}_W$, coincides with the authors' own EP in all relevant examples given in that source. We note the following salient properties, again omitting the proofs.

Proposition 7. $\mathrel{\sim\!\!\!\!\!\rightarrow}_O$ *satisfies CP, Blind Rejection and Base Logic Invariance.*

Proposition 8. $\mathrel{\sim\!\!\!\!\!\rightarrow}_O$ *satisfies Closed Order Compliance.*

Proposition 9. $\mathrel{\sim\!\!\!\!\!\rightarrow}_O$ *is sharp.*

Proposition 10. $\mathrel{\sim\!\!\!\!\!\rightarrow}_O$ *is the weakest EP satisfying CP and Order Compliance.*

Corollary 2. $\mathrel{\sim\!\!\!\!\!\rightarrow}_O$ *is the weakest EP satisfying CP, Weak Order Compliance and Blind Rejection.*

7 Progressive Extension Predicates

Propositions 3 and 10 (and the two corollaries) only characterize $\not\mapsto_O$ and $\not\mapsto_W$ in a relative way as the *weakest* EPs satisfying various principles discussed up to this point: assuming these principles, we can say a bit about which subsets of $\mathcal{L}$ are *not* extensions, but less about which *are*. The following principle adds positive import.

Minimal Application. Let δ be minimal in D, and suppose $W \vdash \alpha$ and $E \Diamond \beta$. Then $(D, W, <) \not\mapsto_x E$ if $(D \setminus \{\delta\}, W \cup \{\gamma\}, <) \not\mapsto_x E$.

In words, if the prerequisite of a maximally preferred default follows from the given facts while its justification is consistent with a proposed extension, then this is indeed an extension, provided it is also an extension of the modified theory obtained by removing this default while treating its consequence as fact.

Proposition 11. *$\not\mapsto_O$ and $\not\mapsto_W$ satisfy Minimal Application.*

As it turns out, Minimal Application is not enough to provide absolute characterisations of $\not\mapsto_O$ and $\not\mapsto_W$, even in combination with conditions previously reviewed, but it does narrow the space of EPs further down. It can also be used as the basis for relative characterisations of two additional EPs, this time looking in the other direction, for the *strongest* possible EPs. Interestingly, an EP showing up now, is the one introduced by Delgrande and Schaub [5].

Definition 6. Let $(D, W, <)$ be an ordered default theory.

1. The PS u is **progressive** if, for any δ_0, δ_1 in u, if $\delta_0 < \delta_1$ then δ_0 occurs before δ_1 in u.
2. $(D, W, <) \not\mapsto_{WP} E$ (E is a **weakly obedient, progressive extension**) if $E = Th(W_u)$ for a progressive and weakly obedient PS u.
3. $(D, W, <) \not\mapsto_{OP} E$ (E is an **obedient, progressive extension**) if $E = Th(W_u)$ for a progressive and obedient PS u.

The following simple example captures the difference between the weaker EPs $\not\mapsto_W$ and $\not\mapsto_O$ and the stronger $\not\mapsto_{WP}$ and $\not\mapsto_{OP}$.

Example 4. Let D contain

$$\delta_1 = p : q/q$$
$$\delta_2 = \top : p/p$$

where $\delta_1 < \delta_2$. Then $(D, \emptyset)$ has the classical extension $Th(\{p, q\})$, obtained by the (only!) PS $\delta_2\delta_1$, which is obedient but not progressive in $(D, W, <)$. Hence $Th(\{p, q\})$ is an extension according to $\not\mapsto_W$ and $\not\mapsto_O$, but not according to $\not\mapsto_{WP}$ or $\not\mapsto_{OP}$.

Simply put, the difference between $\not\mapsto_{WP}$ and $\not\mapsto_{OP}$ is the same as the difference between $\not\mapsto_W$ and $\not\mapsto_O$; it is the difference between Empty Inclusion and Blind Rejection. The following is easily checked:

Proposition 12. $\sim\!\!\!\rightarrow_{WP}$ *satisfies Empty Inclusion.* $\sim\!\!\!\rightarrow_{OP}$ *satisfies Blind Rejection.*

Lemma 3. $(D, W, <) \sim\!\!\!\rightarrow_{OP} E$ *iff* $E = Th(W_u)$ *for some full, progressive and obedient PS* u*.*

The conditions in Lemma 3 are recognized as those used by Delgrande and Schaub [5] in their definition of *order-preserving extension*,[15] hence by happy coincidence the initials in $\sim\!\!\!\rightarrow_{OP}$ may also stand for "order preserving." Again, propositions are listed without proof:

Proposition 13. $\sim\!\!\!\rightarrow_{OP}$ *and* $\sim\!\!\!\rightarrow_{WP}$ *are sharp, and satisfy CP and Minimal Application.*

Proposition 14. $\sim\!\!\!\rightarrow_{WP}$ *satisfies Weak Order Compliance.*

Proposition 15. $\sim\!\!\!\rightarrow_{OP}$ *satisfies Closed Order Compliance.*

Proposition 16. $\sim\!\!\!\rightarrow_{WP}$ *is the strongest EP satisfying CP, Empty Inclusion and Minimal Application.*

Proposition 17. $\sim\!\!\!\rightarrow_{OP}$ *is the strongest EP satisfying CP and Minimal Application.*

Writing "ECP" for **extended core principles**, meaning CP plus Weak Order Compliance and Minimal Application, we sum up the essential findings so far in Figure 1. Finally, we observe that $\sim\!\!\!\rightarrow_{WP}$, but not $\sim\!\!\!\rightarrow_{OP}$, satisfies the logical invariance principles discussed at the outset.

Proposition 18. $\sim\!\!\!\rightarrow_{WP}$ *satisfies Base Logic Invariance.*

Proposition 19. $\sim\!\!\!\rightarrow_{OP}$ *violates Trivial Invariance.*

Proof. Let D contain $\delta_1 = p/p$ and $\delta_2 = \top : p/p$, where $\delta_1 < \delta_2$. Then $(D, \emptyset)$ has the classical extension $Th(\{p\})$, given by the proof sequences δ_2 and $\delta_2\delta_1$, of which the first is not obedient and the second not progressive in $(D, W, <)$. If the trivial default δ_1 is deleted from D, however, δ_2 becomes an obedient, progressive PS. □

Observe that $\sim\!\!\!\rightarrow_{BE}$ and $\sim\!\!\!\rightarrow_{OP}$ violate Base Logic Invariance in directly opposite ways; trivial defaults represent *obstacles* for $\sim\!\!\!\rightarrow_{OP}$, making it more difficult to obtain given extensions, while $\sim\!\!\!\rightarrow_{BE}$ exploit them for "cheating" – obtaining extensions that would not be there without strategically placed (highly prioritised) trivial defaults. One could argue that $\sim\!\!\!\rightarrow_{BE}$ and $\sim\!\!\!\rightarrow_{OP}$ are too weak and strong,

[15] See Definition 4.2 in that reference. Recall that present discussions confine themselves to theories with finite sets of defaults. When E is a classical extension of (D, W) then a grounded enumeration of $GD(D, E)$ is the same as a full PS for E. Condition (1) says that this full PS is progressive, and condition (2) says that it is obedient.

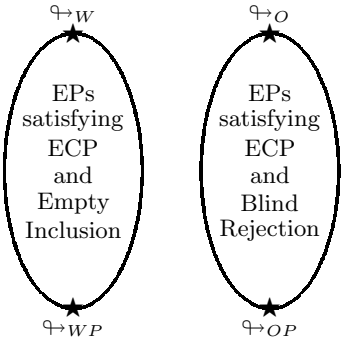


Fig. 1. The ovals denote the classes of EPs satisfying ECP, along with Empty Inclusion and Blind Rejection, respectively. By Proposition 4, the two sets are disjoint. The stars at the top and bottom denote, respectively, the weakest and strongest predicates in the sets. The figure remains true if "sharp" is added to any of the ovals, if "Closed Order Compliance" is added to the rightmost oval, and if "Base Logic Invariance" is added to the leftmost. What is not apparent from the picture, is that $\hookrightarrow_{OP}$ is stronger than $\hookrightarrow_{WP}$ and thus also stronger than $\hookrightarrow_W$, while $\hookrightarrow_O$ is stronger than $\hookrightarrow_W$, and $\hookrightarrow_O$ and $\hookrightarrow_{WP}$ are incomparable.

respectively, to satisfy Base Logic Invariance, and that reasonable repairs would yield $\hookrightarrow_W$ and $\hookrightarrow_{WP}$, respectively.

Schaub and Wang [13] introduced a notion of preferred answer sets for extended logic programs with preferences which, when we seek to "translate" it to the present setting of default logics, would seem to yield an EP strictly between $\hookrightarrow_{OP}$ and $\hookrightarrow_{WP}$: write $(D, W, <) \hookrightarrow_{SW} E$ iff there exists a progressive PS u for E satisfying the following requirement, which (when applied to progressive proof sequences) falls between Obedience and Weak Obedience.

$$W_r \vdash \gamma_0 \text{ if } \delta_0 < \delta_1 \text{ and } r\delta_1 \preceq u \text{ and } W_u \vdash \alpha_0 \text{ and } W_r \Diamond \beta_0.$$

It can be seen that this yields an EP strictly between $\hookrightarrow_{OP}$ and $\hookrightarrow_{WP}$. Since we suggest that $\hookrightarrow_{WP}$ is a reasonable repair of $\hookrightarrow_{OP}$ securing Base Logic Invariance, it is of interest to see how this intermediate notion fares with respect to Base Logic Invariance. A quick look at the example in the proof of Proposition 19, however, immediately makes it clear that also $\hookrightarrow_{SW}$ violates Trivial (and hence also Base Logic) Invariance.

8 Further Directions

In future work we should like to see additional principles capturing intuitions about the preference order which may, in combination with principles discussed here, provide exact characterisations of the various EPs. Questions about which, if any, distinctions explored in this paper disappear when restricting attention to normal defaults and other special cases, also deserve further attention.

Acknowledgement

The work reported on in this paper grew out of joint work with Iselin Engan, Espen Lian and Arild Waaler in Engan et al. [7]. The author is particularly indebted to Iselin Engan for fruitful discussions concerning the interpretation of ordered default theories.

References

1. Baader, F., Hollunder, B.: Priorities on Defaults with Prerequisites, and Their Application in Treating Specificity in Terminological Default Logic. J. Autom. Reasoning 15(1), 41–68 (1995)
2. Brewka, G.: Adding Priorities and Specificity to Default Logic. In: MacNish, C., Moniz Pereira, L., Pearce, D.J. (eds.) JELIA 1994. LNCS, vol. 838, pp. 247–260. Springer, Heidelberg (1994)
3. Brewka, G., Eiter, T.: Preferred Answer Sets for Extended Logic Programs. Artificial Intelligence 109(1-2), 297–356 (1999)
4. Brewka, G., Eiter, T.: Prioritizing default logic. In: Intellectics and Computational Logic: Papers in Honor of Wolfgang Bibel, pp. 27–45. Kluwer, Dordrecht (2000)
5. Delgrande, J.P., Schaub, T.: Expressing preferences in default logic. Artificial Intelligence 123(1-2), 41–87 (2000)
6. Engan, I.: Reasoning with preference in only knowing logic with confidence levels. Master's thesis, University of Oslo, Department of Linguistics, Norway (August 2005)
7. Engan, I., Langholm, T., Lian, E., Waaler, A.: Default reasoning with preference within only knowing logic. In: Baral, C., Greco, G., Leone, N., Terracina, G. (eds.) LPNMR 2005. LNCS, vol. 3662, pp. 304–316. Springer, Heidelberg (2005)
8. Horty, J.: Defaults with Priorities. Journal of Philosophical Logic 36, 367–413 (2007)
9. Langholm, T.: Default Logics with Preference Order: Principles and Characterisations. Extended version of this paper, with mathematical proofs, http://www.sksk.no/langholm/publications/princhar.pdf
10. Marek, V.W., Truszczynski, M.: Nonmonotonic Logic: Context-Dependent Reasoning. Springer, Heidelberg (1993)
11. Reiter, R.: A logic for default reasoning. Artificial Intelligence 13, 81–132 (1980)
12. Rintanen, J.: On Specificity in Default Logic. In: Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI 1995), pp. 1474–1479. Morgan Kaufmann, San Francisco (1995)
13. Schaub, T., Wang, K.: A semantic framework for preference handling in answer set programming. Theory and Practice of Logic Programming 3, 569–607 (2003)
14. Tarski, A.: Logic, Semantics, Metamathematics. Oxford University Press, Oxford (1956)

On Computing Constraint Abduction Answers

Michael Maher and Ge Huang

NICTA and University of NSW
Sydney, Australia
Michael.Maher@nicta.com.au,
plutohg@gmail.com

Abstract. We address the problem of computing and representing answers of constraint abduction problems over the Herbrand domain. This problem is of interest when performing type inference involving generalized algebraic data types. We show that simply recognizing a maximally general answer or fully maximal answer is co-NP complete. However we present an algorithm that computes the (finite) set of fully maximal answers of an abduction problem. The maximally general answers are generally infinite in number but we show how to generate a finite representation of them when only unary function symbols are present.

1 Introduction

Constraint abduction is the inference procedure that, given constraints B and C, infers constraint A such that $A \wedge B \rightarrow C$. Recent work on constraint-based type inference for generalized algebraic data types (GADTs) [15] has used conjunctions of expressions $B \rightarrow C$ to express the type requirements of a program [11,13]. Answers to a constraint abduction problem correspond to a well-typing of the program [13,14]. Some approaches to type inference with GADTs require programmer annotations [9,8], while others attempt to infer types without such help [13,14]. In this paper we explore representational and computational issues that arise in the latter approach, by addressing them in the general context of constraint abduction.

In the Herbrand constraint domain the maximally general answers represent all answers, but there are in general infinitely many maximally general answers to a constraint abduction problem instance [3]. Thus computing them is not straightforward. Furthermore, we show that simply recognising that an answer is maximally general is a co-NP complete problem. There are two ways we can address this problem. The first approach is to develop a representation scheme whereby the set of maximally general answers can be finitely presented. This is a difficult problem and we obtain a solution only in the case where all function symbols are unary.

The second approach is to find a finite subset of the maximally general answers that is canonically defined and of use in practice. The class of fully maximal answers was identified in [3] as omitting many "unexpected" and unhelpful maximally general answers, and was shown to be finite. It is used in [13,14]. Here we provide an algorithm that applies without restriction on the function symbols and generates all fully maximal answers.

For both maximally general answers and fully maximal answers we address specifically the case where there are no function symbols. We have simple characterizations

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 421–435, 2008.

in this case, and we use them to show that the number of maximally general answers and fully maximal answers can grow explosively, even in this simple case.

After some preliminaries on constraint abduction and the Herbrand constraint domain (Section 2) we address, in turn, the complexity of recognising answers (Section 3), and the problems of representing all maximally general answers (Section 4) and computing all fully maximal answers (Section 5).

2 Background

The syntax and semantics of constraints are defined by a constraint domain. Given a signature Σ, and a set of variables $Vars$ (which we assume is infinite), a *constraint domain* is a pair $(\mathcal{D}, \mathcal{L})$ where $\mathcal{D}$ is a Σ-structure and $\mathcal{L}$ (the language of constraints) is a set of Σ-formulas closed under conjunction and renaming of free variables.

Constraint abduction is a form of abduction where all predicates in the formulas over which the abduction is being inferred have a semantics determined by a constraint domain.

Definition 1. *The* Simple Constraint Abduction (SCA) Problem *is as follows:*

Given a constraint domain $(\mathcal{D}, \mathcal{L})$, and given two constraints $B, C \in \mathcal{L}$ such that $\mathcal{D} \models \tilde{\exists}\, B \wedge C$, for what constraints $A \in \mathcal{L}$ does

$$\mathcal{D} \models (A \wedge B) \rightarrow C$$

and

$$\mathcal{D} \models \tilde{\exists}\, (A \wedge B)$$

An instance *of the problem has a fixed constraint domain and fixed constraints B and C. A constraint A satisfying the above properties is called an* answer.

Intuitively, B is background information and C is a conclusion drawn from B and some missing information; each answer A is a candidate for the missing information. Throughout this paper, A, B and C refer to the constraints in a simple constraint abduction problem. We assume $B \wedge C$ is satisfiable; otherwise, there are no answers. Usually we leave the constraint domain implicit and use $(A \wedge B) \rightarrow C$, for example, in place of $\mathcal{D} \models (A \wedge B) \rightarrow C$.

Of all the answers, we are most interested in the *maximally general answers*, that is, constraints A such that $(A \wedge B) \rightarrow C$ and for every A', if $A \rightarrow A'$ and $(A' \wedge B) \rightarrow C$ then $A' \rightarrow A$. (That is, there is no answer strictly more general than A.) Under some conditions on the constraint domain, they can also be thought of as a means to (somewhat) compactly represent all answers.

In general, we wish to solve several SCA problems simultaneously, but it is shown in [3] that maximally general answers to such joint problem can be constructed from the maximally general answers of the individual SCA problems using the algorithm JCA-Solve. Furthermore, if C is $c_1 \wedge \cdots \wedge c_n$ then we can reduce the SCA problem involving B and C to the problem of solving simultaneously the SCA problems involving B and c_i, for $i = 1, \ldots, n$. Thus we can assume without loss of generality that C consists of a single constraint.

In general, there can be infinitely many maximally general answers to a SCA problem, many of which result in a conjunction $A \wedge B$ that is *not* maximally general. In

contexts where A will later be combined with B, we might want $A \wedge B$ to be maximally general. This has led to the definition of fully maximal answers [3] – a subset of the maximally general answers that is finite in some constraint domains.

Definition 2. *An answer A is* fully maximal *if A is a maximally general answer and $A \wedge B$ is maximally general among all expressions $A' \wedge B$ where A' is an answer.*

Equivalently, A is a fully maximal answer iff A is a maximally general answer and $(A \wedge B) \leftrightarrow (B \wedge C)$.

In this paper we are primarily interested in the Herbrand constraint domain, which consists of (possibly) existentially quantified conjunctions of equations on terms, where equality of ground terms is syntactic identity. We will denote this constraint domain by $\mathcal{FT}_\exists$. The weaker constraint domain where existential quantifiers are not used is denoted by $\mathcal{FT}$. These domains are widely used for symbolic computation in automated reasoning, logic programming and type systems. Unification [10,2] is an algorithm for solving equations in these constraint domains.

We assume that in every SCA instance there is a finite set of variables $\tilde{x}$ of interest. Usually we can take $\tilde{x}$ to be $vars(B) \cup vars(C)$, where $vars(o)$ denotes the free variables of o. A constraint in $\mathcal{FT}_\exists$ is in *standard form* if it has the form $\exists \tilde{y}\ \tilde{x} = \tilde{t}(\tilde{y})$, where $\tilde{x}$ is a sequence of all variables of interest and $\tilde{y}$ is a disjoint set of existentially quantified variables. The righthandside of such a constraint in standard form is the sequence of terms $\tilde{t}(\tilde{y})$. Every satisfiable constraint in $\mathcal{FT}_\exists$ (and $\mathcal{FT}$) can be presented in this form. In $\mathcal{FT}$ a constraint is in *solved form* if it has the form $\tilde{x} = \tilde{t}(\tilde{y})$, where $\tilde{x}$ and $\tilde{y}$ are disjoint.

Example 1. We consider a SCA problem instance over $\mathcal{FT}$. Let B be $k(h(x), y) = k(v, g(z))$ and C be $v = h(f(z))$. The solved form of B is $v = h(x), y = g(z)$. Among the maximally general answers to this SCA instance are: the trivial answer $v = h(f(z))$; $x = f(z)$; and $x = f(u), y = g(u)$, for any variable u other than v, x, y, z. The latter class of answers are among the "unexpected" maximally general answers [3] to this instance; they involve a variable not involved in the problem. The answers in this class are not fully maximal. The other two answers are fully maximal. For example, $v = h(f(z)) \wedge v = h(x) \rightarrow x = f(z)$.

If we consider this problem as an instance over $\mathcal{FT}_\exists$ then the standard form of B is $\exists u_1, u_2\ v = h(u_1), x = u_1, y = g(u_2), z = u_2$ and the standard form of C is $\exists u_3\ v = h(f(u_3)), z = u_3$. Again $v = h(f(z))$ and $x = f(z)$ are maximally general answers. Another fully maximal answer is $\exists u\ x = f(u), y = g(u)$, which is strictly more general than $x = f(u), y = g(u)$ for any variable u.

In the Herbrand constraint domains, whether $\mathcal{FT}_\exists$ or $\mathcal{FT}$, the maximally general answers of a SCA problem instance represent all answers, and in general there are infinitely many of them, but always only a finite number of fully maximal answers [3].

3 Recognising Answers

It is straightforward to determine whether a given constraint A is an answer for a SCA problem involving B and C. It can be done in time linear in the size of A, B, C,

using a linear unification algorithm [7,5]. However, determining whether A is maximally general or fully maximal is substantially harder.

Theorem 1. *Let A, B, and C be constraints of $\mathcal{FT}$, and consider the SCA involving B and C.*

1. *The question whether A is an answer can be decided in linear time*
2. *Recognising that A is a maximally general answer is co-NP complete*
3. *Recognising that A is a fully maximal answer is co-NP complete*

To prove parts 2 and 3 we reduce SAT to the problem of finding a more general answer than a given answer. We conjecture that in $\mathcal{FT}_\exists$ recognising that an answer is maximally general or fully maximal is also co-NP complete. However Theorem 1 and its proof do not directly extend to $\mathcal{FT}_\exists$.

4 Maximally General Answers

In this section we consider only Herbrand domains where the signature Σ contains only constants and unary function symbols. The restriction to unary function symbols does not limit the unruly proliferation of maximally general answers as identified in [3].

Example 2. Let B be $x_0 = x_1, x_2 = x_3$ and C be $v = z$. Then, in addition to the more obvious maximally general answers is A_y defined by $x_0 = s(v), x_1 = s(y), x_2 = t(y), x_3 = t(z)$ for any variable y not occurring in B or C, and any terms s and t.

Before defining an algorithm we must introduce several definitions. Let Σ_1 be the set of unary function symbols in Σ. We will use words constructed from these symbols to represent the repeated application of the functions. For example, the application of functions $f(g(h(f(x))))$ is represented by the word $fghf$ applied to the variable x. The application of a word w to a term t is written $w(t)$. The empty word is denoted by ε.

While function symbols represent tree (or term) constructors, we introduce inverse elements as the deconstructors for the underlying function symbol. Thus f^{-1} applied to the term $f(g(x))$, or $f^{-1}(f(g(x)))$, is equal to $g(x)$. In terms of equations, $x = f^{-1}(y)$ is defined to mean $y = f(x)$. We extend the inverse notation to general expressions by defining $(uw)^{-1} = w^{-1}u^{-1}$. In some cases the application of an inverse element to a term is not meaningful, for example, application of g^{-1} to $f(g(x))$. It corresponds to a clash of function symbols in unification. As a result, composition of expressions is a partial function. For example, $(ff^{-1})g$ is equivalent to g, but $f(f^{-1}g)$ represents a clash.

We can formulate equational reasoning in a partial algebra $\mathcal{W}$ of term constructors and destructors where the values are Σ_1-words, ε is the empty word, each $\sigma \in \Sigma_1$ is a constant, there is a binary composition operator (represented by juxtaposition) and the inverse operator $^{-1}$. (The problem of solving equations on this partial algebra has some similarity to solving equations on the free group with rational constraints forcing each variable to be a word [1].)

The meaning of expressions in the algebra is given by formulas in $\mathcal{FT}_\exists$ relating two variables, and similarly for equations over the algebra. Let e, e_1, e_2 be expressions in the algebra and w be a Σ_1-word.

$$\begin{aligned}
[\![x\ (w)\ y]\!] &\text{ is the equation } x = w(y)\\
[\![x\ (e^{-1})\ y]\!] &= [\![y\ (e)\ x]\!]\\
[\![x\ (e_1 e_2)\ y]\!] &= \exists z\ [\![x\ (e_1)\ z]\!] \wedge [\![z\ (e_2)\ y]\!]\\
[\![e_1 = e_2]\!] &= \forall x, y\ ([\![x\ (e_1)\ y]\!] \leftrightarrow [\![x\ (e_2)\ y]\!])
\end{aligned}$$

We introduce an infinite set $WVars$ of variables ranging over words, and extend expressions to incorporate word variables. A *word expression* is an expression in the algebra that does not involve the inverse operator. Clearly, the composition operator is associative in cases where both results are defined. We adopt the convention that composition associates to the right. For a set S of expressions, we define $S^{-1} = \{s^{-1} \mid s \in S\}$.

Recall that we write $B \rightarrow c$ for $\mathcal{FT} \models B \rightarrow c$. We say terms s and t are B-*equivalent* if $B \rightarrow s = t$. We write $[s]$ for the B-equivalence class of s.

We group the terms that are (perhaps indirectly) equationally related by B. Given a set of equations B, a B-*class* is a minimal non-empty set S of terms that is closed under (a) B-equivalence, (b) taking subterms, and (c) taking superterms. Each B-class has at least one term (variable or constant) such that neither it, nor any term B-equivalent to it, has a subterm. For each B-class we fix one such term and refer to it as the *base* of the B-class. We say a B-class has a *constant base* if it contains a constant (in which case we will choose the constant to be the base without loss of generality). Every term in a B-class is B-equivalent to a term with the base as a subterm. Thus, for every term t there is a (unique) corresponding word w_t such that $B \rightarrow t = w_t(b)$, where b is the base of the B-class containing t. Note that all B-classes are disjoint and, given a variable or constant z, all terms containing an occurrence of z are in the same B-class. Since we assume B is satisfiable in $\mathcal{FT}$, each B-class contains, at most, one constant.

Given two variables x and y in the same B-class, they are equationally related via $x = w_x(b), y = w_y(b)$, where b is the base of the B-class. We can visualize the parts of the B-class relevant to x and y as shown in Figure 1(a). In those terms, $w_x = ac$ and $w_y = dc$, where c is the greatest common suffix[1] of w_x and w_y.

A B-class contains infinitely many B-equivalence classes, assuming Σ_1 is non-empty. However, a finite representation of the B-classes can be computed by applying a congruence closure algorithm [6] to the equations in B. The constants and variables not congruent to any term with subterms are candidates for the base of a B-class. It is straightforward to use an ordering on variables and constants to canonically choose a base. We will assume that a constant is chosen as base, whenever possible.

Now, let C be $v = w(z)$, where v is a variable, but z may be a variable or a constant. Consider a prospective answer A, which we can assume is in solved form.

The *coarse* A, B-*graph* is an undirected graph where the B-classes are vertices, for each equation $(s_1 = s_2) \in A$ there is an edge between the B-class of s_1 and the B-class of s_2, and there are no other edges. We can use the coarse A, B-graph to eliminate some

[1] The *greatest common suffix* of words w_1 and w_2 is the longest word w such that there are words u_1 and u_2 with $w_1 = u_1 w$ and $w_2 = u_2 w$.

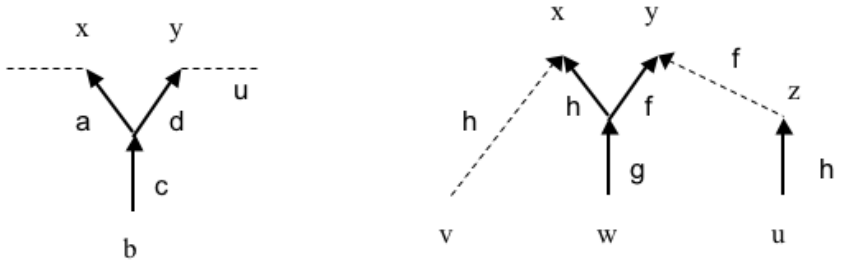


Fig. 1. Diagrams of (a) a B-class, and (b) the relevant part of the A, B-graph for Example 3

prospective answers that are not maximally general answers. A *simple path* between B-classes b_1 and b_2 is a minimal set of edges that connects b_1 and b_2. When necessary, we can arrange the edges in a sequence, to form a path as conventionally defined.

Proposition 1. *Consider a solved form constraint A and its coarse A, B-graph. If A is a maximally general answer of the SCA problem involving B and C ($v = w(z)$) then then the coarse A, B-graph consists exactly of a simple path between the B-class of z and the B-class of v.*

However, the coarse A, B-graph does not address the details of A and cannot even be used to determine whether $A \wedge B$ is satisfiable, much less characterize maximally general answers. Hence we define a more detailed graph.

The A, B*-graph* is a labelled, directed graph where the B-equivalence classes are vertices and there is an edge from $[s]$ to $[t]$ labelled with a word u if either $B \rightarrow t = u(s)$ or $t = u(s) \in A$. A *simple path* in the A, B-graph is a simple path in the underlying undirected graph.

We will use paths in this graph to represent equational reasoning that might be used to infer C (i.e. $v = u(z)$) from $A \wedge B$. Traversing an edge in the direction of an arrow labelled with u corresponds to applying the word u to the terms in the current B-equivalence class, that is, constructing (the B-equivalence class of) larger terms from the current terms. Conversely, traversing an edge *against* the direction of an arrow labelled with u corresponds to applying u^{-1} to the current B-equivalence class and deconstructing the current terms, that is, deleting the prefix u from the word defining a current term to produce (the B-equivalence class of) a subterm. If u is not a prefix of the word defining the term then we cannot establish an equational relationship between the terms at the two endpoints of the edge.

An *MGA-path* from z to v is a sequence of B-equivalence classes $E_0, \ldots, E_{2n+1}$ such that $E_0 = [z]$, $E_{2n+1} = [v]$, E_{2i} and E_{2i+1} are in the same B-class, for $i = 0, \ldots, n$, and the path involves at most one B-class with a constant base. (Note that E_{2i} and E_{2i+1} may be the same B-equivalence class.) The MGA-path is *induced* by the solved form constraint A if $\{E_{2i+1}, E_{2i+2}\} = \{[x], [y]\}$ for some equation $x = u(y) \in A$, for $i = 0, \ldots, n-1$ and, conversely, for each equation $x = u(y) \in A$ we have $\{[x], [y]\} = \{E_{2i+1}, E_{2i+2}\}$, for some i. We say that A and the MGA-path *correspond*. Notice that the edges relevant to the previous definition involve only B-equivalence classes of variables or constants. Hence only finitely many MGA-paths follow the same

route as a simple path in the coarse A, B-graph. An MGA-path is *simple* if the sequence contains either 2 or 0 occurrences of B-equivalence classes from each B-class.

Example 3. Let C be $v = z$, B be $x = hg(w), y = fg(w), z = h(u)$ and A be $x = h(v), y = f(z)$, where u, v, w, x, y, z are variables. Then there are three non-simple B-classes: one contains v as the base, and is non-simple only because v occurs in C; one contains z and u, with u as the base; and one contains w, x and y, with w as the base. Part of the A, B-graph is shown in Figure 1(b). The MGA-path induced by A is the path from $[z]$ to $[v]$ via $[x]$ and $[y]$. The coarse A, B-graph consists of the three non-simple B-classes, connected by the dashed edges.

We now refine Proposition 1.

Proposition 2. *A maximally general answer A induces a simple MGA-path from $[z]$ to $[v]$ in the A, B-graph.*

Since the equational relationship between any two B-equivalence classes $[x]$ and $[y]$ in the same B-class is as described in Figure 1(a), a maximally general answer must induce a path within the A, B-graph of the form shown in Figure 1(b), where the dashed edges correspond to equations in A. Let us index the variables and words. We will use the naming scheme for variables and words in a B-class described in Figure 1(a) with an index i for the i'th B-class (counting from v, on the left), so that $B \rightarrow (x_i = a_i c_i(b_i) \wedge y_i = d_i c_i(b_i))$ and equations in A relate y_i and x_{i+1}. These equations may be of the form $y_i = u_i(x_{i+1})$ or $x_{i+1} = u_i(y_i)$, where u_i is a word. We express these two possibilities compactly as $y_i = u_i^{e_i}(x_{i+1})$ where $e_i \in \{1, -1\}$.

We use the algebra of term constructors and their inverses to formulate requirements on A to be a maximally general answer. The main requirement is the condition for A, represented by a path from z to v, to establish that $v = w(z)$.

$$a_1 d_1^{-1} u_1^{e_1} \ldots a_i d_i^{-1} u_i^{e_i} \ldots a_n d_n^{-1} = w \tag{1}$$

This is a necessary, but not sufficient, condition for A to be a maximally general answer.

Example 4. Continuing with Example 3 in Figure 1(b), the path from $[z]$ to $[v]$ via $[y]$ and $[x]$ is labelled $h^{-1}hf^{-1}f$, which is equal to ε, implying that $A \wedge B \rightarrow C$. However $A \wedge B$ is inconsistent: there is a clash between the h in $z = h(u)$ and the g in $y = fg(w)$. Thus A is not an answer. If, in place of $z = h(u)$, B contained $z = a$ then we would have a clash between a and g.

The problem is that, while the MGA-path demonstrates the possibility that we have a maximally general answer, a $c_i(b_i)$ (in terms of Figure 1) can be incompatible with either another c_j, as in the above example, or with part of the MGA-path. There are two possibilities: a clash between a constant and a function symbol, or a clash between function symbols.

The first possibility arises only if the path contains a B-class with a constant base b. Suppose it is the m'th B-class. For every simple path from b to another base we require that the result is a word. This ensures that there is no clash between b and a unary

function symbol. There are two variations of the constraint, depending on whether the target base is in a B-class between b and z or between b and v.

For each $j : m < j \leq n$

$$\exists u \;\; c_j^{-1} a_j^{-1} u_j^{-e_j} d_{j-1} \ldots a_{i+1}^{-1} u_i^{-e_i} d_i \ldots d_{m+1} a_{m+1}^{-1} u_m^{-e_m} d_m c_m = u \tag{2}$$

For each $j : 1 \leq j < m$

$$\exists u \;\; c_j^{-1} d_j^{-1} u_j^{e_j} a_{j+1} \ldots a_i d_i^{-1} u_i^{e_i} \ldots a_{m-1} d_{m-1}^{-1} u_{m-1}^{e_{m-1}} a_m c_m = u \tag{3}$$

If there is a clash between function symbols on the MGA-path the requirement (1) will not be satisfied, because the left side will not evaluate to a word. We need to examine all paths between bases to ensure that there is no clash between the function symbols off the MGA-path and any other function symbol. Every variable-free word expression without a clash can be simplified to the form $u'u^{-1}$, for some words u and u'. Hence we require that each path expression between bases will evaluate to this form.

For each $j, m : 1 \leq j < m \leq n$

$$\exists u, u' \;\; c_j^{-1} d_j^{-1} u_j^{e_j} a_{j+1} \ldots a_i d_i^{-1} u_i^{e_i} \ldots a_{m-1} d_{m-1}^{-1} u_{m-1}^{e_{m-1}} a_m c_m = u' u^{-1} \tag{4}$$

The requirements (1) – (4) are necessary and sufficient for A to be a maximally general answer.

Theorem 2. *Consider a SCA problem involving B and C ($v = w(z)$) and a solved form constraint A. A is a maximally general answer if and only if A corresponds to a simple MGA-path in the A, B-graph from $[z]$ to $[v]$, and, for that path, requirements (1) – (4) are satisfied.*

Thus if we can finitely represent all solutions to the requirements (1) and (2) – (4), where we now regard the u_i's as variables, then we also represent all maximally general answers. This leads us to the abstract algorithm in Figure 2. Note that the output equations may contain (possibly constrained) word variables in the solutions for u_i's; any consistent instantiation of all these variables by words gives a maximally general answer.

algorithm MGA(B, C)
  for $n = 1, \ldots, m$ **do**
    for every usable sequence of B-classes $B_1, \ldots, B_n$ **do**
      for $i = 1, \ldots, n-1$ **do**
        choose value of e_i
        choose values for $[y_i]$ in B_i and $[x_{i+1}]$ in B_{i+1} to form a MGA-path
      Generate equations (1) - (4) in variables u_i
      Solve equations for u_i
      choose variable or constant representatives p_i and q_i for each $[x_i]$ and $[y_i]$
      Check that the output describes a sufficiently general solved form
      output equations $q_i = u_i^{e_i}(p_i)$
end algorithm

Fig. 2. Nondeterministic algorithm for maximally general answers

For this approach to produce a finite representation, we must have an upper bound m on the number of B-classes in a sequence. A B-class is *simple* if it contains exactly one variable or constant (which must be the base) and does not contain v or z. A variable (constant) that appears in a simple class is called a *simple variable* (*simple constant*). It turns out that no maximally general answer A in solved form can correspond to a MGA-path with two adjacent simple B-classes because of the syntactic form of solved forms and the lack of any other term B-equivalent to the simple variable or constant. Hence if q is the number of non-simple B-classes, then we can take $m = 2q - 1$; no maximally general answer corresponds to a longer MGA-path. A sequence of B-classes in *usable* if simple B-classes do not appear consecutively in the sequence. Unusable sequences do not correspond to maximally general answers.

There is one further restriction on simple variables in maximally general answers: Simple variables must appear on the righthandside of equations in A, but must not appear as a bare variable (i.e. they must appear as part of a larger term). For example, if x_s is a simple variable then $y_1 = s(x_s), y_2 = t(x_s)$ is acceptable as part of A, assuming s and t are non-empty words, but $y_1 = x_s, y_2 = t(x_s)$ is less general than $y_2 = t(y_1)$ and, similarly, $x_s = s(y_1), y_2 = t(x_s)$ is less general than $y_2 = ts(y_1)$, while $x_s = s(y_1), x_s = t(y_2)$ is not in solved form. The same point applies to simple constants. Although non-simple variables may appear on the lefthandside of equations in A, they also may not appear as a bare variable on the righthandside (except for v and z), nor may they appear both on the lefthandside and the righthandside of equations in A. For simplicity, we express these restrictions in the algorithm of Figure 2 as a check before output, but clearly it would be more efficient to enforce them at the time choices are made.

The names of simple variables are unimportant: any renaming of these variables will result in a different maximally general answer. Thus any maximally general answer involving a simple variable other than v and z represents an infinite set of maximally general answers (since $Vars$ is infinite). The answers involving simple variables are some of the "unexpected" maximally general answers discussed in [3] and Example 1.

Rather than solve the equations (1) – (4) directly, we employ a backtracking search procedure that produces a finite representation of the values of the word variables u_i for which the equations A on an MGA-path form a maximally general answer. There is no room in this paper to present the complete algorithm *solve*, but we give an outline.

The state of the algorithm is described by the following parameters: the remainder of the MGA-path to be explored; a variable or constant b that is the deepest base of a B-class discovered so far; word expressions r, s, and t describing the relationship between the current point p, the desired value of v (that is, $w(z)$) and b; and constraints E on word variables. The relationship of some of these parameters is displayed in Figure 3.

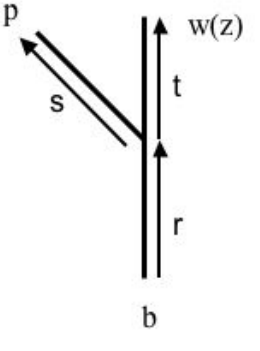


Fig. 3. Diagram of algorithm parameters

In the algorithm, we iterate along expression $a_1 d_1^{-1} u_1^{e_1} \ldots a_i d_i^{-1} u_i^{e_i} \ldots a_n d_n^{-1}$ of (1) from right to left, preserving the invariants listed below at each point p (in B-class B_j) of that expression by updating the parameters. The algorithm branches non-deterministically when different updates are possible, and on each branch accumulates conditions on the word variables u_i. These conditions E are output when the branch terminates successfully. If E is unsatisfiable then the branch fails (i.e. terminates unsuccessfully). Thus the algorithm has a constraint programming style.

Let N be the number of symbols (variables and constants) in (1). Each branch has length bounded by N. At each point the branching factor is at most $N + n$. Testing satisfiability of E terminates, since the constraints involved have a restricted form. Hence the algorithm must terminate.

Let e denote the part of the expression already visited. The invariants are:

1. the longest common suffix of s and t is ε
2. $A \wedge B \models_E ts^{-1}(p) = tr(b) = w(z)$
3. $A \wedge B \models_E p = sr(b) = e(z)$
4. b is a constant iff some B_i has a constant base for $j \leq i \leq n$
5. $A \wedge B \models_E (2)_j \wedge (3)_j \wedge (4)_j$

Here $A \wedge B \models_E \psi$ denotes that for every valuation σ of the word variables that satisfies E, $\mathcal{FT} \models \sigma(A) \wedge B \rightarrow \sigma(\psi)$ (where $\sigma(A)$ and $\sigma(\psi)$ are well-formed (sets of) equations in the language of $\mathcal{FT}$). $(i)_j$ denotes the subset of equations (i) that refers to paths within and between B-classes $B_j, B_{j+1}, \ldots, B_n$.

The collection of conditions E output by the algorithm *solve* constitute a finite representation of the solutions of equations (1) – (4).

Theorem 3. *Consider a SCA problem involving B and C over $\mathcal{FT}$, and equations (1) – (4) for a given MGA-path.*

1. *The algorithm* solve *terminates.*
2. *Let σ be a solution to the equations (1) – (4). Then σ can be extended to a solution of one of the outputs of the* solve *algorithm.*
3. *If E is an output of the* solve *algorithm then every solution of E, when restricted to the free variables of equations (1) – (4), is a solution of those equations.*

Combining Theorem 2 and Theorem 3, we establish that the algorithm *MGA*(B, C) in Figure 2 produces a finite representation of all maximally general answers of a simple constraint abduction problem over $\mathcal{FT}$ where C is a single equation. When C contains multiple equations we apply the JCA-Solve algorithm of [3] to the outputs of *MGA*(B, c), for each $c \in C$, as mentioned earlier.

Example 5. Consider a variation of Example 3 where C is $v = z$ and B is $x = hg(w), y = fg(w)$. The B-classes of v, y, z form a usable sequence and there is a MGA-path $[z]$, $[y]$, $[x]$, $[v]$. Of the possible values of the exponents, only one combination leads to equations with a solution. We consider this case, where A will have the form $x = u_1(v), y = u_2(z)$.

The resulting constraints on the word variables u_i are $u_1^{-1} f h^{-1} u_2 = \varepsilon$ from (1) and the following three constraints from (4): $\exists u, u'\ g^{-1} h^{-1} u_2 = u' u^{-1}$, $\exists u, u'\ u_1^{-1} f h^{-1} u_2 = u' u^{-1}$, which is redundant wrt the first equation, and $\exists u, u'\ u_1^{-1} f g = u' u^{-1}$.

The *solve* algorithm produces the following two descriptions of solutions to the constraints: $u_1 = f, u_2 = h$ and $u_1 = fgu, u_2 = hgu$ for any word u. These correspond to the answers $x = f(v), y = h(z)$ and $x = f(g(k(v))), y = h(g(k(z)))$ for any term k (including the empty term). There are, of course, answers derived from other MGA-paths.

Only small modifications are need to handle $\mathcal{FT}_\exists$ constraints: B-classes might contain existential variables, and these cannot be used as representatives of classes in a MGA-path, and simple variables must be considered existential variables.

A Signature of Constants

When there are only constants in the signature the above discussion becomes much simpler. There is no distinction between $\mathcal{FT}$ and $\mathcal{FT}_\exists$, the B-classes reduce to B-equivalence classes, simple B-classes become irrelevant, the A, B-graph is identical to the coarse A, B-graph, and all labels on edges of the the A, B-graph are the empty word. Thus every solved form constraint that corresponds to a simple path from $[z]$ to $[v]$ is a maximally general answer.

We can use this to count the number of maximally general answers in some cases. For example,

Proposition 3. *Consider a SCA problem where the signature Σ consists only of constants, C is $v = z$ and at most one constant appears in B and C. The number of maximally general answers is*

$$\sum_{n=2}^{q+2} \sum_{\text{sequences } B_1 \ldots B_n} |B_1| * |B_n| * \prod_{i=1}^{n-1} |B_i| * (|B_i| - 1)$$

where q is the number of non-simple B-classes, $|B_i|$ denotes the cardinality of a B-class B_i, and we sum over all sequences of non-simple B-classes with $B_1 = [z]$ and $B_n = [v]$.

When there are function symbols the number of maximally general answers can grow rapidly – doubly exponentially – or can be infinite [3]. But even for SCA problems that do not involve function symbols, the number of maximally general answers can grow very rapidly.

Example 6. Consider the SCA problem where C is $x_1 = x_{2m}$ and B is $x_1 = x_2, \ldots, x_{2i-1} = x_{2i}, \ldots, x_{2m-1} = x_{2m}$. Then $|B_i| = 2$, for $i = 1, ..., m$ and $|B_1| * |B_n| * \prod_{i=2}^{n} |B_i| * (|B_i| - 1) = 2^n$ for any n. For any length n there are $(n-2)!$ sequences that start with B_0 and end with B_n. Thus there are $\sum_{n=2}^{m} (n-2)! * 2^n$ maximally general answers. If $m \geq 2$ then there is a lower bound of $(m-2)!2^m$ and an upper bound of $(m-1)!2^m$.

Note that this discussion only applies when the signature does not contain any unary (or higher arity) function symbols. It is not sufficient to simply impose that the SCA is function-free (i.e. no such function symbols appear in B or C).

Example 7. Let B be $u = v$ and C be $x = y$. Then the algorithm in Figure 2 generates the maximally general answers $x = y$, and $x = u, y = v$, and $x = v, y = u$. However, if the signature contains function symbols then there are also maximally general answers $x = k(u), y = k(v)$ and $v = k(x), u = k(y)$, among others, for any term k.

Nevertheless, it seems that Example 6 provides a lower bound on the growth of the number of expressions needed to represent all maximally general answers in the worst case. Example 6 is also suggestive of the number of maximally general answers for *any* constraint domain containing equations, since the problem and the answers are valid in any constraint domain. On the other hand, there are some (non-equational) constraint domains that have a unique most general answer [4].

5 Fully Maximal Answers

To a very limited extent we can use the results of the previous section to compute fully maximal answers. For example

Proposition 4. *Consider a SCA problem over a signature of constants. Suppose C consists of a single equation $v = z$. Then the fully maximal answers are those defined by the algorithm MGA with a sequence of B-classes of length 2. Thus there are $|[v]| * |[z]|$ fully maximal answers.*

When C contains more than one equation the JCA-Solve algorithm will not necessarily produce all the fully maximal answers from the individual fully maximal answers.

Example 8. Let B be $x = y, z = w$ and let C be $x = a, z = a$. Then the smaller SCA problems where c_1 is $x = a$ (and c_2 is $z = a$) have fully maximal answers $x = a$ and $y = a$ (respectively, $z = a$ and $w = a$). The JCA-Solve algorithm combines these answers to give four fully maximal answers (such as $x = a, z = a$).

However, there are other fully maximal answers such as $x = z, y = a$ that are not composed from fully maximal answers to smaller problems.

Thus we take a more direct approach to computing fully maximal answers. Fortunately, under this approach we need no restriction on the signature. We will first address the problem over the constraint domain $\mathcal{FT}_\exists$, and later discuss the small modifications needed to adapt it to $\mathcal{FT}$.

Suppose A is a constraint of $\mathcal{FT}_\exists$ in standard form and S is a nonempty set of positions in A. We define a few auxiliary functions.

$pos(t, A)$ returns the set of positions of the term t in the righthandside of A
$repl(S, A)$ replaces all terms in the righthandside of A occurring at a position in S by a new variable (that is existentially quantified)
$next(A)$ is the set $\{repl(S, A) \mid S$ is a nonempty set of positions of identical terms t in A such that $t \notin Vars,$ or $S \subset pos(t, A)\}$

$next(A)$ is a set of constraints strictly more general than A. All constraints (up to equivalence) more general than A can be generated by iterating $next$.

algorithm FMA(B, C)
  if $(B \rightarrow C)$ **then return** $true$
  let A be the standard form of $B \wedge C$
  do forever
    let $\mathcal{A}$ be $next(A)$
    if $(\forall A' \in \mathcal{A}\ A' \wedge B \not\rightarrow C)$ **then return** A
    choose $A \in \mathcal{A}$ such that $A \wedge B \rightarrow C$
end algorithm

Fig. 4. Nondeterministic algorithm for computing fully maximal answers

The algorithm defined in Figure 4 non-deterministically finds a maximally general answer that is more general than $B \wedge C$. Using backtracking to implement the non-determinism, we can enumerate all fully maximal answers.

It is straightforward to see that the algorithm is correct and terminates: By the conditions in the definition of $next$, each iteration of the loop makes A strictly more general. Since for each constraint there are only finitely many more general constraints, the loop must terminate. An invariant of the loop is that A is an answer and is more general than $B \wedge C$. When the loop is exited, there are no answers more general than A, and hence A must be a maximally general answer. Thus, Definition 2, A is fully maximal. Furthermore, since any constraint that is more general than $B \wedge C$ can be obtained from $B \wedge C$ by a sequence of $next$ operations, every fully maximal answer is generated. Thus

Theorem 4. *Algorithm FMA outputs all fully maximal answers to the SCA problem over $\mathcal{FT}_\exists$ and terminates.*

There are optimizations that could be applied to the algorithm. In particular, it will find the same fully maximal answer several times because the order in which subterms t of A are generalized by $repl$ is not significant to the final outcome. If we restrict the order in which subterms t are chosen we will avoid this possibility, and restrict the branching factor.

We can also require that subterms t that are chosen do not contain new variables (introduced by $repl$). This prevents the algorithm proceeding by several "partial" generalizations.

Another possibility is to require an incremental approach, where $next(A)$ contains only constraints minimally (strictly) more general than A. That can be achieved by restricting t in the definition of $next$ to terms containing at most one function symbol.

We need vary the algorithm only slightly to find fully maximal answers over the constraint domain $\mathcal{FT}$. We assume A is in solved form, and redefine two auxiliary functions as follows.

$repl(S, A, x, t)$ deletes $x = t$ from A and replaces all occurrences of t at a position in S by x
$next(A)$ is the set $\{repl(S, A, x, t) \mid x = t \in A, S \subseteq pos(t, A)\}$

A variant of this algorithm has been proposed by Tom Schrijvers [12]. The correctness of the algorithm follows from the same argument as for the previous theorem, underpinned by results from [2] on the structure of constraints in $\mathcal{FT}$.

Theorem 5. *Algorithm FMA, with the auxiliary functions modified as above, outputs all fully maximal answers to the SCA problem over $\mathcal{FT}$ and terminates.*

The correctness of the algorithm relies only on $next(A)$ returning all constraints minimally more general than A and termination relies only on every constraint having only finitely many more general constraints. Thus the abstract algorithm in Figure 4 can be adapted to *any* constraint domain where a constraint has only finitely many generalizations, provided $next$ can be defined constructively.

Obviously these algorithms have high complexity. To some extent this cannot be avoided. [3] has an example with a binary function symbol where the number of fully maximal answers in $\mathcal{FT}$ grows doubly exponentially with the size of B and C. Again we look at the function-free case, where we have a more direct characterization of the fully maximal answers.

A Signature of Constants

When Σ contains only constants we can consider the A, B-graph as an undirected graph. We say A connects B-classes $[s]$ and $[t]$ if there is a path from $[s]$ to $[t]$ in the A, B-graph. We now characterize the fully maximal answers in the function-free case.

Theorem 6. *Consider a function-free SCA problem. A is a fully maximal answer iff A and C connect the same B-equivalence classes and the A, B-graph is a forest.*

Unlike the computation of maximally general answers discussed in Section 4, this result permits signatures containing function symbols (provided those symbols do not appear in B or C). Using this characterization we can show that the number of fully maximal answers can grow exponentially.

Example 9. Suppose C is $y_i = t_i$ for $i = 1, \ldots, n$, where each y_i and t_i occurs just once in C, and B consists of the equations $y_i = y'_i$ for $i = 1, \ldots, n$ where the y'_i are variables not appearing elsewhere in B and C. Then the size of the problem, combining B and C, is $\theta(n)$.

The connected subgraphs of the C, B-graph all consist of a single edge connecting two vertices (or the trivial case of an isolated vertex). It follows from Theorem 6 that any fully maximal answer A has an A, B-graph isomorphic to the C, B-graph. The i'th edge might be represented in A by $y_i = t_i$ or $y'_i = t_i$. Thus there are 2^n inequivalent fully maximal answers to this problem.

We can make the same point here as in the discussion of maximally general answers over a signature of constants: it appears that most constraint domains involving equations will have a similar growth in the number of fully maximal answers.

6 Conclusion

We have investigated constraint abduction over the Herbrand domain. We have shown how to compute fully maximal answers, and represent finitely all maximally general

answers in the unary case. However, these problems are intractable in their full generality, and even in terms of the number of fully maximal answers in the function-free case. This suggests that the use of constraint abduction for practical type inference is out of reach until a smaller subset of answers can be identified with the meaning of the type constraints, or more compact representations can be found. For the latter quest, the A, B-graph is a starting point.

Acknowledgements. The authors thank J. Jaffar, T. Schrijvers and P. Stuckey for discussions related to this paper. We thank the referees for their comments, which helped improve the presentation. NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

References

1. Diekert, V., Gutierrez, C., Hagenah, C.: The existential theory of equations with rational constraints in free groups is PSPACE-complete. Information and Computation 202(2), 105–140 (2005)
2. Lassez, J.-L., Maher, M.J., Marriott, K.G.: Unification Revisited. In: Minker, J. (ed.) Foundations of Deductive Databases and Logic Programming, pp. 587–625. Kauffman, San Francisco (1987)
3. Maher, M.J.: Herbrand Constraint Abduction. In: Proc. Symp. on Logic in Computer Science, pp. 397–406 (2005)
4. Maher, M.: Heyting Domains for Constraint Abduction. In: Sattar, A., Kang, B.-h. (eds.) AI 2006. LNCS (LNAI), vol. 4304, pp. 9–18. Springer, Heidelberg (2006)
5. Martelli, A., Montanari, U.: An Efficient Unification Algorithm. ACM Trans. Program. Lang. Syst. 4(2), 258–282 (1982)
6. Nelson, G., Oppen, D.C.: Fast Decision Procedures Based on Congruence Closure. JACM 27(2), 356–364 (1980)
7. Paterson, M., Wegman, M.N.: Linear Unification. J. Comput. Syst. Sci. 16(2), 158–167 (1978)
8. Peyton Jones, S., Vytiniotis, D., Weirich, S., Washburn, G.: Simple Unification-based Type Inference for GADTs. In: Proc. Int. Conf. Functional Programming, pp. 50–61. ACM Press, New York (2006)
9. Pottier, F., Régis-Gianas, Y.: Stratified type inference for generalized algebraic data types. In: Proc. POPL, pp. 232–244. ACM Press, New York (2006)
10. Robinson, J.A.: A Machine-Oriented Logic Based on the Resolution Principle. JACM 12(1), 23–41 (1965)
11. Simonet, V., Pottier, F.: Constraint-based type inference with guarded algebraic data types. ACM Transactions on Programming Languages and Systems 29(1) (2007)
12. Schrijvers, T.: Personal communication
13. Stuckey, P.J., Sulzmann, M., Wazny, J.: Type Processing by Constraint Reasoning. In: Kobayashi, N. (ed.) APLAS 2006. LNCS, vol. 4279, pp. 1–25. Springer, Heidelberg (2006)
14. Sulzmann, M., Schrijvers, T., Stuckey, P.J.: Type inference for GADTs via Herbrand constraint abduction, Report CW 507, K.U.Leuven, Dept. of Computer Science (2008)
15. Xi, H., Chen, C., Chen, G.: Guarded recursive datatype constructors. In: Proc. POPL, pp. 224–235 (2003)

Fast Counting with Bounded Treewidth*

Michael Jakl, Reinhard Pichler, Stefan Rümmele, and Stefan Woltran

Vienna University of Technology

Abstract. Many intractable problems have been shown to become tractable if the treewidth of the underlying structure is bounded by a constant. An important tool for deriving such results is Courcelle's Theorem, which states that all properties defined by Monadic-Second Order (MSO) sentences are fixed-parameter tractable with respect to the treewidth. Arnborg et al. extended this result to counting problems defined via MSO properties. However, the MSO description of a problem is of course not an algorithm. Consequently, proving the fixed-parameter tractability of some problem via Courcelle's Theorem can be considered as the starting point rather than the endpoint of the search for an efficient algorithm. Gottlob et al. have recently presented a new approach via monadic datalog to actually devise efficient algorithms for decision problems whose tractability follows from Courcelle's Theorem. In this paper, we extend this approach and apply it to some fundamental counting problems in logic an artificial intelligence.

1 Introduction

Many problems which are, in general, intractable, have been shown to become tractable if the treewidth of the underlying structure is bounded by a constant. An important tool for deriving such results is Courcelle's Theorem [1]. It states that any property of finite structures, which is expressible by a Monadic Second Order (MSO) sentence, can be decided in linear time (data complexity) if the structures under consideration have bounded treewidth. Courcelle's Theorem has been successfully applied to derive tractability results in a great variety of fields. Recently, also its applicability to AI has been underlined by showing that many fundamental problems in the area of nonmonotonic reasoning and knowledge representation can be encoded as MSO sentences [2]. In [3], it was shown that the fixed-parameter tractability (FPT) via Courcelle's Theorem can be extended to counting problems defined via MSO properties.

Clearly, the MSO description of a problem is not an algorithm. Previous methods for constructing concrete algorithms from an MSO description [3,4] first transform the MSO evaluation problem into a tree language recognition problem, which is then solved via a finite tree automaton (FTA). However, this approach has turned out to be only of theoretical value, since even very simple MSO formulae quickly lead to a "state explosion" of the FTA (see [5]). Consequently, it was already stated in [6] that the algorithms derived via Courcelle's Theorem are "useless for practical applications" and that the main benefit of Courcelle's Theorem is in providing "a simple way to recognize a property as being linear time computable". Of course, this also applies to the extension of Courcelle's Theorem to counting problems according to [3]. In other words, proving the FPT of some problem by showing that it is MSO expressible is the starting point rather than the end point of the search for an efficient algorithm.

* This work was supported by the Austrian Science Fund (FWF), project P20704-N18.

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 436–450, 2008.

Recently, an alternative method to tackle this class of fixed-parameter tractable problems via *monadic datalog* has been proposed in [7]. In particular, it has been shown that if some property of finite structures is expressible in MSO then it can also be expressed by means of a monadic datalog program over the structure plus the tree decomposition. The monadic datalog approach has been applied to problems from different areas [7,8] including propositional satisfiability (SAT) and abduction. In this paper, we show that the monadic datalog approach can be extended in such a way that it also provides concrete algorithms for some fundamental counting problems.

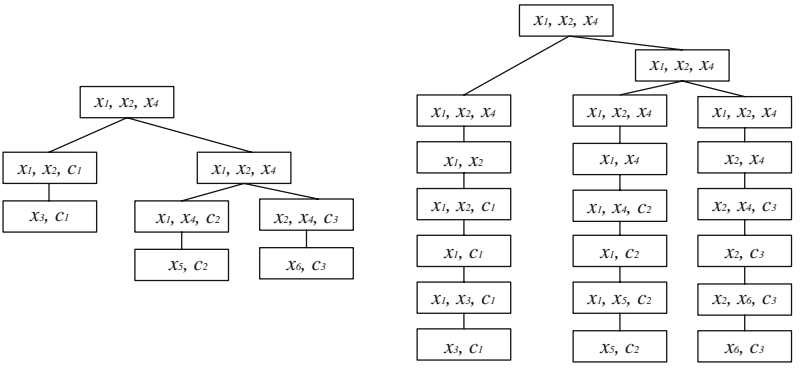
Results. We present new algorithms for the following problems: #SAT – the problem of counting *all* models of a propositional formula (without restriction, this is a classical #P-complete problem); #CIRCUMSCRIPTION – the problem of counting the (subset) *minimal* models of a propositional formula (this problem was, apart from the generic $\#\Pi_1$SAT-problem, one of the first problems to be shown #NP-complete [9]); and #HORN-ABDUCTION – the problem of counting the solutions of a propositional abduction problem where the underlying theory is given by a set of Horn clauses. The #P-completeness of this problem has been recently shown in [10]. Finally, we also report on experimental evaluations of the #SAT algorithm. In particular, we compare a dedicated implementation (where datalog serves as a "specification") with direct realizations of the datalog approach on top of the DLV-system [11]. Our experiments underline that our approach of counting indeed yields the expected fixed-parameter tractability and that – in great contrast to the MSO-to-FTA approach – there are no "hidden constants" in the runtime behavior to render these algorithms useless.

Related Work. As mentioned above, counting problems defined via MSO properties were shown in [3] to be FPT w.r.t. the treewidth of the input structures. In [12], this FPT result was extended to graphs with bounded clique-width. An algorithm for solving #SAT and #GENSAT in case of bounded treewidth or clique-width of the primal or incidence graph was presented in [13]. Moreover, it is sketched how this approach based on recursive splitting can be extended to other #P-complete problems. In [14], new #SAT-algorithms based on dynamic programming were presented for bounded treewidth of several graphs related to a propositional formula in CNF, namely the primal graph, dual graph, and incidence graph. Our notion of treewidth of a CNF-formula (see Section 2) corresponds to the treewidth of the incidence graph.

2 Preliminaries

Finite Structures and Treewidth. Let $\tau = \{R_1, \ldots, R_K\}$ be a set of predicate symbols. A *finite structure* $\mathcal{A}$ over τ (a *τ-structure*, for short) is given by a finite domain $A = dom(\mathcal{A})$ and relations $R_i^{\mathcal{A}} \subseteq A^\alpha$, where α is the arity of $R_i \in \tau$. A *tree decomposition* $\mathcal{T}$ of a τ-structure $\mathcal{A}$ is a pair $\langle T, (A_t)_{t \in T} \rangle$ where T is a tree and each A_t is a subset of A, s.t. the following properties hold: (1) Every $a \in A$ is contained in some A_t. (2) For every $R_i \in \tau$ and every tuple $(a_1, \ldots, a_\alpha) \in R_i^{\mathcal{A}}$, there exists a node $t \in T$ with $\{a_1, \ldots, a_\alpha\} \subseteq A_t$. (3) For every $a \in A$, $\{t \mid a \in A_t\}$ induces a subtree of T.

The sets A_t are called the *bags* of $\mathcal{T}$. The *width* of a tree decomposition $\langle T, (A_t)_{t \in T} \rangle$ is defined as $max\{|A_t| \mid t \in T\} - 1$. The *treewidth* of $\mathcal{A}$ is the minimal width of all tree decompositions of $\mathcal{A}$. It is denoted as $tw(\mathcal{A})$. For given $w \geq 1$, it can be decided in linear time if some structure has treewidth $\leq w$. Moreover, in case of a positive answer, a tree decomposition of width w can be computed in linear time [15].



(a) tree decomposition $\mathcal{T}_1$ (b) tree decomposition $\mathcal{T}_2$

Fig. 1. Tree decompositions of formula φ of Example 1

Example 1 ([2]). We can represent propositional formulae in CNF as finite structures over the alphabet $\tau = \{cl(.), var(.), pos(.,.), neg(.,.)\}$ where $cl(z)$ (resp. $var(z)$) means that z is a clause (resp. a variable) and $pos(x, c)$ (resp. $neg(x, c)$) means that x occurs unnegated (resp. negated) in the clause c. For instance, the formula $\varphi = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_4 \vee \neg x_5) \wedge (x_2 \vee \neg x_4 \vee x_6)$ corresponds to the structure $\mathcal{A}$ given by the set of ground atoms $\{var(x_1), var(x_2), var(x_3), var(x_4), var(x_5), var(x_6), cl(c_1), cl(c_2), cl(c_3), pos(x_1, c_1), pos(x_3, c_1), pos(x_4, c_2), pos(x_2, c_3), pos(x_6, c_3), neg(x_2, c_1), neg(x_1, c_2), neg(x_5, c_2), neg(x_4, c_3)\}$. Two tree decompositions $\mathcal{T}_1$ and $\mathcal{T}_2$ of $\mathcal{A}$ are given in Figure 1. Note that the maximal size of the bags is 3 in both decompositions. Hence, the treewidth is ≤ 2. On the other hand, it can be shown that these tree decompositions are optimal in the sense that we have $tw(\varphi) = tw(\mathcal{A}) = 2$. $\diamond$

In [7], it was shown that the following form of *normalized tree decompositions* can be obtained in linear time: (1) All bags contain either w or $w + 1$ pairwise distinct elements. W.l.o.g., we may assume that the domain contains at least w elements. (2) Every internal node $t \in T$ has either 1 or 2 child nodes. (3) If a node t has one child node t', then the bag A_t is obtained from $A_{t'}$ either by removing one element or by introducing a new element. (4) If a node t has two child nodes then these child nodes have identical bags as t. In this case, we call t a *branch node*. In this paper, we only deal with finite structures representing propositional formulae in CNF (possibly Horn). Hence, the domain elements are either variables or clauses. Consequently, in case (3), we call a node t in the tree decomposition a *variable removal node*, a *clause removal node*, a *variable introduction node*, or a *clause introduction node*, respectively.

The tree decomposition $\mathcal{T}_2$ in Figure 1 is normalized in this sense.

MSO and Monadic Datalog. MSO extends First Order logic (FO) by the use of *set variables* (denoted by upper case letters), which range over sets of domain elements. In contrast, the *individual variables* (denoted by lower case letters) range over single domain elements. An MSO formula $\varphi(x)$ with exactly one free individual variable is called a *unary query*. *Datalog* programs are function-free logic programs. The (minimal-model) semantics can be defined as the least fixpoint (lfp) of applying the immediate consequence operator. Predicates occurring only in the body of rules are called extensional. Predicates occurring also in the head of some rule are called intensional.

Let $\mathcal{A}$ be a τ-structure of treewidth $w \geq 1$. Then we define the extended signature $\tau_{td} = \tau \cup \{root, leaf, child_1, child_2, bag\}$, where the unary predicates $root$ and $leaf$ as well as the binary predicates $child_1$ and $child_2$ are used to represent the tree of a tree decomposition (of width w) in the obvious way. Finally, predicate bag has arity $k+2$ with $k \leq w$, where $bag(t, a_0, \ldots, a_k)$ means that the bag at node t is $(a_0, \ldots, a_k)$.

In [7], the following connection between unary MSO queries over structures with bounded treewidth and monadic datalog was established:

Theorem 1. *Let τ and $w \geq 1$ be arbitrary but fixed. Every MSO-definable unary query over τ-structures of treewidth w is also definable by a monadic datalog program over τ_{td}. Moreover, the resulting program can be evaluated in linear time w.r.t. the size of the original τ-structure.*

3 Counting All Models

We start our investigation of counting problems with the #SAT problem, i.e.: given a clause set $\mathcal{C}$ over variables V, count the number of all models $J \subseteq V$ of $\mathcal{C}$ (We identify an assignment with the set of atoms that are true in it). Suppose that an instance of #SAT is given as a τ_{td}-structure with $\tau_{td} = \{cl, var, pos, neg, root, leaf, child_1, child_2, bag\}$, encoding a clause set together with a tree decomposition $\mathcal{T}$ of width w (as explained Example 1). An extended datalog program for #SAT is displayed in Figure 2.

In this program, we adhere to the following notational conventions: Lower case letters v, c, x, and j (possibly with subscripts) are used as datalog variables for a single node in $\mathcal{T}$, for a single clause, for a single propositional variable, or for an integer number, respectively. Upper case letters are used as datalog variables denoting sets of variables (in the case of X, P, N) or sets of clauses (in the case of C). In particular, for the sake of readability, we present the extensional predicate bag in the form $bag(v, X, C)$, where X (resp. C) denotes the set of variables (resp. clauses) in the bag at node v in $\mathcal{T}$. Note that all these sets are not sets in the general sense, since their cardinality is restricted by the maximal size $w+1$ of the bags, where w is a fixed constant. Indeed, we ultimately feed these sets to the datalog system DLV in the form of individual arguments of appropriate variants of the predicates involved, see Section 6.

We are also using non-datalog expressions involving the $\cup$- and $\uplus$-operator for ordinary resp. disjoint union. They could be easily replaced by "proper" datalog expressions, e.g., $C_1 \cup C_2 = C_u$ can of course be replaced by union(C_1, C_2, C_u). Moreover, we need arithmetic expressions $j_1 + j_2$ and $j_1 * j_2$ as well as the SUM-operator for the counting. The SUM-operator occurs as the expression SUM(j) in the rule heads only. Its semantics is like the SUM aggregate function in ordinary SQL, where we first apply a GROUP BY over all remaining head variables to the result of evaluating the conjunctive query in the body of the rule.

For the discussion of the #SAT program below, it is convenient to introduce the following notation: Let $\mathcal{C}$ denote the input clause set with variables in V and tree decomposition $\mathcal{T}$. For any node v in $\mathcal{T}$, we write $\mathcal{T}_v$ to denote the subtree of $\mathcal{T}$ rooted at v. By $Cl(v)$ we denote the clauses in the bag of v while $Cl(\mathcal{T}_v)$ denotes the clauses that occur in any bag in $\mathcal{T}_v$. Analogously, we write $Var(v)$ and $Var(\mathcal{T}_v)$ as a short-hand for the variables occurring in the bag of v respectively in any bag in $\mathcal{T}_v$. Finally, the restriction of a clause c to the variables in some set $U \subseteq V$ will be denoted by $c|_U$.

Program #SAT

/* leaf node. */
$sat(v, P, N, C_u, 1) \leftarrow leaf(v), bag(v, X, C), partition(X, P, N), true(P, N, C_u, C).$

/* variable removal node. */
$sat(v, P, N, C_u, j_1 + j_2) \leftarrow bag(v, X, C), child_1(v_1, v), bag(v_1, X \uplus \{x\}, C),$
$\quad sat(v_1, P \uplus \{x\}, N, C_u, j_1), sat(v_1, P, N \uplus \{x\}, C_u, j_2).$
$sat(v, P, N, C_u, j) \leftarrow bag(v, X, C), child_1(v_1, v), bag(v_1, X \uplus \{x\}, C),$
$\quad sat(v_1, P \uplus \{x\}, N, C_u, j),$ not $sat(v_1, P, N \uplus \{x\}, C_u, \_).$
$sat(v, P, N, C_u, j) \leftarrow bag(v, X, C), child_1(v_1, v), bag(v_1, X \uplus \{x\}, C),$
$\quad sat(v_1, P, N \uplus \{x\}, C_u, j),$ not $sat(v_1, P \uplus \{x\}, N, C_u, \_).$

/* clause removal node. */
$sat(v, P, N, C_u, j) \leftarrow bag(v, X, C), child_1(v_1, v), bag(v_1, X, C \uplus \{c\}),$
$\quad sat(v_1, P, N, C_u \uplus \{c\}, j).$

/* variable introduction node. */
$sat(v, P \uplus \{x\}, N, C_u, \text{SUM}(j)) \leftarrow bag(v, X \uplus \{x\}, C), child_1(v_1, v), bag(v_1, X, C),$
$\quad sat(v_1, P, N, C_1, j), true(\{x\}, \emptyset, C_2, C), C_1 \cup C_2 = C_u.$
$sat(v, P, N \uplus \{x\}, C_u, \text{SUM}(j)) \leftarrow bag(v, X \uplus \{x\}, C), child_1(v_1, v), bag(v_1, X, C),$
$\quad sat(v_1, P, N, C_1, j), true(\emptyset, \{x\}, C_2, C), C_1 \cup C_2 = C_u.$

/* clause introduction node. */
$sat(v, P, N, C_u, j) \leftarrow bag(v, X, C \uplus \{c\}), child_1(v_1, v), bag(v_1, X, C),$
$\quad sat(v_1, P, N, C_1, j), true(P, N, C_2, \{c\}), C_1 \cup C_2 = C_u.$

/* branch node. */
$sat(v, P, N, C_u, \text{SUM}(j)) \leftarrow child_1(v_1, v), bag(v_1, X, C), sat(v_1, P, N, C_1, j_1),$
$\quad child_2(v_2, v), bag(v_2, X, C), sat(v_2, P, N, C_2, j_2),$
$\quad bag(v, X, C), C_1 \cup C_2 = C_u, j_1 * j_2 = j.$

/* result (at the root node). */
$count(\text{SUM}(j)) \leftarrow root(v), bag(v, X, C), sat(v, P, N, C, j).$

Fig. 2. #SAT program

The #SAT program contains four intensional predicates sat, $true$, $partition$, and $count$. The crucial predicate is $sat(v, P, N, C, j)$ with the following intended meaning: v denotes a node in $\mathcal{T}$. P and N form a partition of $Var(v)$ representing a truth assignment on $Var(v)$, s.t. all variables in P are true and all variables in N are false. C denotes a subset of $Cl(v)$ and j denotes a positive integer. For arbitrary values of v, P, N, C, we define the following set of truth assignments:

$$S(v, P, N, C) = \{J \mid J \text{ is an extension of } (P, N) \text{ to } Var(\mathcal{T}_v), \text{ for each } c \in (Cl(\mathcal{T}_v) \setminus Cl(v)) \cup C, c \text{ is true in } J, \text{ for each } c \in Cl(v) \setminus C, c|_{Var(\mathcal{T}_v)} \text{ is false in } J. \}$$

We can now characterize the least fixpoint (lfp) of the #SAT program as follows.[1]

Property A. If $S(v, P, N, C) = \emptyset$ then no atom $sat(v, P, N, C, \_)$ is in the lfp of #SAT. If $S(v, P, N, C) \neq \emptyset$ then the following equivalence holds: $sat(v, P, N, C, j)$ is in the lfp of #SAT iff $|S(v, P, N, C)| = j$.

[1] Due to lack of space, most proofs are omitted in this paper (for full proofs, we refer to [16]).

This property implies that, for any given values v, P, N, C, we can derive at most one fact $sat(v, P, N, C, j)$. The main task of the program is the computation of all facts $sat(v, P, N, C, j)$ by means of a bottom-up traversal of the tree decomposition $\mathcal{T}$. Indeed, all the rules only allow us to derive sat-facts for some node v in $\mathcal{T}$ from sat-facts at the child node(s) of v. Consequently, on the ground level, the program contains only stratified negation, since the not-operator in the rules of variable removal nodes is only applied to sat-facts of the child node v_1 of v.

The other predicates have the following meaning: $true(P, N, C_u, C)$ means that C_u contains precisely those clauses from C which are true in the (partial) assignment given by (P, N). We do not specify the implementation of this predicate here. It can be easily achieved via the extensional predicates pos and neg. A fact $partition(X, P, N)$ expresses that (P, N) is a partition of X. The predicate $count$ holds the final result. The datalog program in Figure 2 solves the #SAT problem in the following way.

Theorem 2. *Let $\mathcal{C}$ be an instance of #SAT, encoded by a τ_{td}-structure $\mathcal{A}_{td}$. Then, $count(j)$ with $j \geq 1$ is in the lfp of the #SAT-program evaluated on $\mathcal{A}_{td}$ iff $\mathcal{C}$ is satisfiable and has exactly j models. Moreover, both the construction of the τ_{td}-structure $\mathcal{A}_{td}$ and the evaluation of the program take time $\mathcal{O}(f(tw(\mathcal{C})) * \|\mathcal{C}\|)$ for some function f, if we assume constant runtime for the arithmetic operations.*

Proof. Suppose that the predicate sat indeed fulfills Property A, which can be proved by structural induction on $\mathcal{T}$. The case distinction over all possible kinds of nodes is rather straightforward – the only non-trivial case being the case of branch nodes.

Now consider the root node v of the tree decomposition $\mathcal{T}$ with $bag(v, X, C)$. A fact $sat(v, P, N, C, j)$ in the lfp means that the assignment (P, N) on the variables X has exactly j extensions to all variables, s.t. all clauses in $\mathcal{C}$ are true. But then, by the semantics of the SUM-operator explained above, the rule with head $count(\text{SUM}(j))$ indeed means that a fact $count(j')$ with $j' \geq 1$ is in the lfp iff j' is the number of assignments that satisfy all clauses in $\mathcal{C}$, i.e., j' is the sum of j over all possible partitions (P, N) of X, s.t. $sat(v, P, N, C, j)$ is in the lfp. We are thus using that the root v of $\mathcal{T}$ is unique and the values of X and C in the bag at v are uniquely determined by v. Moreover, for every pair of (P, N) (together with C, which is fixed for v), the value of j is also uniquely determined.

The linear time data complexity is due to the fact that our #SAT program is essentially a succinct representation of a monadic datalog program extended by a counter j. For instance, in the atom $sat(v, P, N, C, j)$, the sets P, N, and C are subsets of bounded size of the bag of v. Hence, each combination P, N, C could be represented by sets $r, s, t \subseteq \{0, \ldots, w\}$ referring to indices of elements in the bag of v. Recall that w is a fixed constant. Hence, $sat(v, P, N, C, j)$ is simply a succinct representation of constantly many predicates of the form $sat_{r,s,t}(v, j)$. Hence, without the counter j, the linear time bound is implicit in Theorem 1. Moreover, j is uniquely determined for every combination of v, P, N, C, and the concrete value of j is computed by simple addition and multiplication of the corresponding values in sat-facts at the child node(s) of v. Hence, maintaining this additional argument j does not destroy the linearity. □

4 Counting the Minimal Models

We now extend the #SAT program in order to solve the #CIRCUMSCRIPTION problem, i.e.: given a propositional formula φ, count the number of *minimal* models of φ. The

Program #CIRCUMSCRIPTION

/* leaf node. */
$sat(v, 0, P, N, C_u, 1) \leftarrow leaf(v), bag(v, X, C), partition(X, P, N), true(P, N, C_u, C).$
$unsat(v, 0, P, N, P', N', C'_u) \leftarrow leaf(v),$
$\quad sat(v, 0, P, N, \_, 1), sat(v, 0, P', N', C'_u, 1), P' \subset P.$

/* variable removal node. */
$auxsat(v, i, 0, P, N, C_u, j) \leftarrow bag(v, X, C), child_1(v_1, v), bag(v_1, X \uplus \{x\}, C),$
$\quad sat(v_1, i, P \uplus \{x\}, N, C_u, j).$
$auxsat(v, i, 1, P, N, C_u, j) \leftarrow bag(v, X, C), child_1(v_1, v), bag(v_1, X \uplus \{x\}, C),$
$\quad sat(v_1, i, P, N \uplus \{x\}, C_u, j).$
$auxunsat(v, i, 0, P, N, P' \setminus \{x\}, N' \setminus \{x\}, C'_u) \leftarrow bag(v, X, C), child_1(v_1, v),$
$\quad bag(v_1, X \uplus \{x\}, C), unsat(v_1, i, P \uplus \{x\}, N, P', N', C'_u).$
$auxunsat(v, i, 1, P, N, P', N' \setminus \{x\}, C'_u) \leftarrow bag(v, X, C), child_1(v_1, v),$
$\quad bag(v_1, X \uplus \{x\}, C), unsat(v_1, i, P, N \uplus \{x\}, P', N', C'_u).$

/* clause removal node. */
$sat(v, i, P, N, C_u, j) \leftarrow bag(v, X, C), child_1(v_1, v), bag(v_1, X, C \uplus \{c\}),$
$\quad sat(v_1, i, P, N, C_u \uplus \{c\}, j).$
$unsat(v, i, P, N, P', N', C'_u) \leftarrow bag(v, X, C), child_1(v_1, v), bag(v_1, X, C \uplus \{c\}),$
$\quad sat(v_1, i, P, N, C_u \uplus \{c\}, \_), unsat(v_1, i, P, N, P', N', C'_u \uplus \{c\}).$

/* variable introduction node. */
$sat(v, i, P \uplus \{x\}, N, C_1 \cup C_2, j) \leftarrow bag(v, X \uplus \{x\}, C), child_1(v_1, v), bag(v_1, X, C),$
$\quad sat(v_1, i, P, N, C_1, j), true(\{x\}, \emptyset, C_2, C).$
$sat(v, i, P, N \uplus \{x\}, C_1 \cup C_2, j) \leftarrow bag(v, X \uplus \{x\}, C), child_1(v_1, v), bag(v_1, X, C),$
$\quad sat(v_1, i, P, N, C_1, j), true(\emptyset, \{x\}, C_2, C).$
$unsat(v, i, P \uplus \{x\}, N, P' \uplus \{x\}, N', C_1 \cup C_2) \leftarrow bag(v, X \uplus \{x\}, C), child_1(v_1, v),$
$\quad bag(v_1, X, C), unsat(v_1, i, P, N, P', N', C_1), true(\{x\}, \emptyset, C_2, C).$
$unsat(v, i, P \uplus \{x\}, N, P', N' \uplus \{x\}, C_1 \cup C_2) \leftarrow bag(v, X \uplus \{x\}, C), child_1(v_1, v),$
$\quad bag(v_1, X, C), unsat(v_1, i, P, N, P', N', C_1), true(\emptyset, \{x\}, C_2, C).$
$unsat(v, i, P \uplus \{x\}, N, P, N \uplus \{x\}, C_1 \cup C_2) \leftarrow bag(v, X \uplus \{x\}, C), child_1(v_1, v),$
$\quad bag(v_1, X, C), sat(v_1, i, P, N, C_1, \_), true(\emptyset, \{x\}, C_2, C).$
$unsat(v, i, P, N \uplus \{x\}, P', N' \uplus \{x\}, C_1 \cup C_2) \leftarrow bag(v, X \uplus \{x\}, C), child_1(v_1, v),$
$\quad bag(v_1, X, C), unsat(v_1, i, P, N, P', N', C_1), true(\emptyset, \{x\}, C_2, C).$

/* clause introduction node. */
$sat(v, i, P, N, C_1 \cup C_2, j) \leftarrow bag(v, X, C \uplus \{c\}), child_1(v_1, v), bag(v_1, X, C),$
$\quad sat(v_1, i, P, N, C_1, j), true(P, N, C_2, \{c\}).$
$unsat(v, i, P, N, P', N', C_1 \cup C_2) \leftarrow bag(v, X, C \uplus \{c\}), child_1(v_1, v), bag(v_1, X, C),$
$\quad unsat(v_1, i, P, N, P', N', C_1), true(P', N', C_2, \{c\}).$

Fig. 3. #CIRCUMSCRIPTION program

goal of the program in Figure 3 and 4 is, on the one hand, to keep track of *all* models of a formula φ given by the input τ_{td}-structure. This is done by the *sat*-predicate which works essentially as in the #SAT program. However, at the end of the day, we may only count the *minimal* models. Our #CIRCUMSCRIPTION program therefore also contains an *unsat*-predicate, which is used to propagate "unsat"-conditions in the sense that some model J is minimal only if all strictly smaller assignments $J' \subset J$ do not satisfy φ. Recall that we identify an assignment with the set of atoms that are true in it.

Program #CIRCUMSCRIPTION (continued)

/* branch node. */
$auxsat(v, i_1, i_2, P, N, C_1 \cup C_2, j_1 * j_2) \leftarrow bag(v, X, C)$, $child_1(v_1, v)$, $bag(v_1, X, C)$,
$sat(v_1, i_1, P, N, C_1, j_1)$, $child_2(v_2, v)$, $bag(v_2, X, C)$, $sat(v_2, i_2, P, N, C_2, j_2)$.
$auxunsat(v, i_1, i_2, P, N, P', N', C_1 \cup C_2) \leftarrow bag(v, X, C)$,
$child_1(v_1, v)$, $bag(v_1, X, C)$, $unsat(v_1, i_1, P, N, P', N', C_1)$,
$child_2(v_2, v)$, $bag(v_2, X, C)$, $unsat(v_2, i_2, P, N, P', N', C_2)$.
$auxunsat(v, i_1, i_2, P, N, P, N, C_1 \cup C_2) \leftarrow bag(v, X, C)$,
$child_1(v_1, v)$, $bag(v_1, X, C)$, $sat(v_1, i_1, P, N, C_1, \_)$,
$child_2(v_2, v)$, $bag(v_2, X, C)$, $unsat(v_2, i_2, P, N, P, N, C_2)$.
$auxunsat(v, i_1, i_2, P, N, P, N, C_1 \cup C_2) \leftarrow bag(v, X, C)$,
$child_1(v_1, v)$, $bag(v_1, X, C)$, $unsat(v_1, i_1, P, N, P, N, C_1)$,
$child_2(v_2, v)$, $bag(v_2, X, C)$, $sat(v_2, i_2, P, N, C_2, \_)$.

/* variable removal and branch node: $aux \Rightarrow sat$ */
$sat(v, i, P, N, C_u, j) \leftarrow auxsat(v, i_1, i_2, P, N, C_u, \_)$, $reduce(v, P, N, i, i_1, i_2, j)$.
$unsat(v, i, P, N, P', N', C'_u) \leftarrow auxunsat(v, i_1, i_2, P, N, P', N', C'_u)$,
$reduce(v, P, N, i, i_1, i_2, \_)$.

/* result (at the root node). */
$count(\text{SUM}(j)) \leftarrow root(v)$, $bag(v, X, C)$, $sat(v, i, P, N, C, j)$,
not $unsat(v, i, P, N, P', N', C)$.

Fig. 4. #CIRCUMSCRIPTION program

A complication which our program has to overcome is that we have to keep track which unsat-conditions refer to which sat-condition. Thus the *sat*-predicate has an index $i \in \{0, 1, 2, \dots\}$ as additional argument. The first four arguments v, i, P, N allow us to associate each *unsat*-fact with the correct *sat*-fact. The *sat*- and *unsat*-predicates have the following meaning: Let v denote a node in the tree decomposition $\mathcal{T}$. Let the sets P and N (resp. P' and N') denote a partition of $Var(v)$ representing a truth assignment on $Var(v)$, s.t. all variables in P (resp. in P') are true and all variables in N (resp. in N') are false. Let C and C' denote subsets of $Cl(v)$. Furthermore let $i \in \{0, 1, 2, \dots\}$ be an index used to distinguish different extensions of a truth assignment and let j be a positive integer used for counting extensions of a truth assignment. Moreover, let the set $S(v, P, N, C)$ of truth assignments be defined as in Section 3. Then, occurrences of the ground facts $sat(v, i, P, N, C, j)$ and $unsat(v, i, P, N, P', N', C')$ in the least fixpoint (lfp) of #CIRCUMSCRIPTION are determined as follows:

Property B. There exists an atom $sat(v, \_, P, N, C, \_)$ in the lfp of the #CIRCUMSCRIPTION program iff $S(v, P, N, C) \neq \emptyset$. Moreover, a fact $unsat(v, i, P, N, \_, \_, \_, \_)$ is in the lfp only if also a fact $sat(v, i, P, N, \_, \_)$ is. Finally, if $S(v, P, N, C) \neq \emptyset$ then there exists a partition $\{S_{i_1}, \dots, S_{i_n}\}$ with $n \geq 1$ of $S(v, P, N, C)$ which fulfills the following conditions:

1. A fact $sat(v, i, P, N, C, \_)$ is contained in the lfp iff $i \in \{i_1, \dots, i_n\}$.
2. The fact $sat(v, i, P, N, C, j)$ is contained in the lfp iff $|S_i| = j$.
3. For every partition (P', N') of $Var(v)$ and every subset $C' \subseteq Cl(v)$, the following two equivalences hold:

The fact $unsat(v, i, P, N, P', N', C')$ is contained in the lfp $\Leftrightarrow$
there exists a $J \in S_i$ and an assignment $J' \subset J$, s.t. $J' \in S(v, P', N', C') \Leftrightarrow$
for all $J \in S_i$ there exists an assignment $J' \subset J$, s.t. $J' \in S(v, P', N', C')$.

Condition 2 above implies that, for any values v, i, P, N, C, there is at most one fact $sat(v, i, P, N, C, \_)$ in the lfp. Condition 3 ensures that, at the root node v of $\mathcal{T}$, either *all* j models described by a fact $sat(v, i, P, N, C, j)$ are minimal or *none* of them is.

The predicates *true* and *partition* have the same meaning as in the #SAT program. In addition, we have the predicates *auxsat*, *auxunsat*, and *reduce* with the following meaning: Recall that the index i in $sat(v, i, P, N, C, \_)$ is used to keep different assignments $J \in S(v, P, N, C)$ apart. Of course, in principle, there can be exponentially many such J. Nonetheless, the predicates *auxsat*, *auxunsat*, and *reduce* guarantee the fixed-parameter tractability in the following way. In the first place, we compute facts $auxsat(v, i_1, i_2, P, N, C, \_)$ and $auxunsat(v, i_1, i_2, P, N, P', N', C')$, where we use *pairs of indices* (i_1, i_2) rather than a single index i to associate the *auxunsat*-facts with the correct *auxsat*-fact. Now suppose that for two distinct pairs (i_1, i_2) and (i_1', i_2') a fact $auxsat(v, i_1, i_2, P, N, C, \_)$ and $auxsat(v, i_1', i_2', P, N, C, \_)$ exists in the lfp and, moreover, the *auxunsat*-facts for (v, i_1, i_2, P, N) and (v, i_1', i_2', P, N) are the same, i.e., for indices i, j, let $Val(v, i, j, P, N) = \{(P', N', C') \mid$ there exists a fact $auxunsat(v, i, j, P, N, P', N', C')$ in the lfp $\}$. Then $Val(v, i_1, i_2, P, N) = Val(v, i_1', i_2', P, N)$ holds. Intuitively, this means that the pairs of indices (i_1, i_2) and (i_1', i_2') are not distinguishable by the *sat*- and *unsat*-conditions for this particular combination of (v, P, N). The purpose of the *reduce*-predicate is, in such a situation, to contract (i_1, i_2) and (i_1', i_2') to a single index i and to take care of the actual counting and summation. More precisely, a fact $reduce(v, P, N, i, i_1, i_2, j)$ means that the pair of indices (i_1, i_2) is mapped to the single index i and that j is the sum of all j' in facts $auxsat(v, i_1', i_2', P, N, C, j')$, s.t. (i_1', i_2') is mapped to i. In principle, the *reduce*-predicate predicate can be realized in datalog (see [16]). However, in the long run, an efficient implementation via hash tables inside the datalog processor is clearly preferable. The datalog program in Figure 3 and 4 solves the #CIRCUMSCRIPTION problem in the following way:

Theorem 3. *Let $\mathcal{C}$ be an instance of* #CIRCUMSCRIPTION*, encoded by a τ_{td}-structure $\mathcal{A}_{td}$. Then, $count(j)$ with $j \geq 1$ is in the lfp of the* #CIRCUMSCRIPTION*-program evaluated on $\mathcal{A}_{td}$ iff $\mathcal{C}$ is satisfiable and has exactly j (subset) minimal models. Moreover, both the construction of the τ_{td}-structure $\mathcal{A}_{td}$ and the evaluation of the program take time $\mathcal{O}(f(tw(\mathcal{C})) * \|\mathcal{C}\|)$ for some function f, if we assume constant runtime for the arithmetic operations.*

Proof. The proof is based on essentially the same ideas as the proof of Theorem 2. In particular, the correctness follows easily as soon as the correctness of Property B is established, which can be done by structural induction. The linear time bound is again shown via Theorem 1 and the fact that the arithmetic operations required for the counting do not destroy the linear time data complexity. □

5 Horn Abduction

Abduction is an important method in artificial intelligence and, in particular, in diagnosis. A propositional abduction problem (PAP) is given by a tuple $\mathcal{P} = \langle V, H, M, \mathcal{C} \rangle$,

where V is a finite set of variables, $H \subseteq V$ is the set of hypotheses, $M \subseteq V$ is the set of manifestations and $\mathcal{C}$ is a consistent theory in the form of a propositional clause set. A set $\mathcal{S} \subseteq H$ is a *solution* to $\mathcal{P}$ if $\mathcal{C} \cup \mathcal{S}$ is consistent and $\mathcal{C} \cup \mathcal{S} \models M$ holds.

In [8], the decision problem (i.e., does a given PAP have a solution) of propositional abduction with bounded treewidth was considered. In order to illustrate the wide applicability of the datalog approach, we concentrate on the special case of #HORN-ABDUCTION, i.e., given a PAP $\mathcal{P}$ whose theory is a set of Horn clauses, count the number of solutions $\mathcal{S}$ of $\mathcal{P}$. The datalog program in Figure 5 has a significantly different flavour than the ones in the previous sections and can be considered as prototypical for rule-based problems.

Before we explain this program, we introduce some useful terminology and conventions: In general, Horn clauses are either rules, facts, or goals. For our purposes, it is convenient to consider every clause r of $\mathcal{C}$ as a *rule* consisting of a head (denoted as $head(r)$) and a body (denoted as $body(r)$). Goals of the form $\neg p_1 \vee \cdots \vee \neg p_k$ are thus considered as rules of the form $p_1 \wedge \cdots \wedge p_k \rightarrow \bot$ and a fact q in $\mathcal{C}$ is considered as a rule of the form $\rightarrow q$ with an empty body. A PAP is represented by a τ-structure with $\tau = \{cl, var, neg, pos, hyp, man\}$, where the predicates hyp and man indicate that some variable a is a hypothesis (i.e., $hyp(a)$) or a manifestation (i.e., $man(a)$). By the above consideration, $var(\bot)$ is now also fulfilled. Moreover, $neg(a, r)$ (resp. $pos(a, r)$) means that a occurs in the body of r (resp. in the head of r). For the input tree decomposition, we assume that a bag containing some rule r also contains the variable a in the head of r. This will greatly simplify the presentation of our datalog program and can, in the worst-case, only double the width of the resulting decomposition.

For $\mathcal{S} \subseteq V \cup \{\bot\}$, we write $\mathcal{S}^+$ to denote the *closure* of $\mathcal{S}$ w.r.t. the theory $\mathcal{C}$, i.e.: An element $q \in V \cup \{\bot\}$ is contained in $\mathcal{S}^+$ iff either $q \in \mathcal{S}$ or there exists a "derivation sequence" of q from $\mathcal{S}$ in $\mathcal{C}$ of the form $\mathcal{S} \rightarrow \mathcal{S} \cup \{q_1\} \rightarrow \mathcal{S} \cup \{q_1, q_2\} \rightarrow \ldots \rightarrow \mathcal{S} \cup \{q_1, \ldots, q_n\}$, s.t. $q_n = q$ and for every $i \in \{1, \ldots, n\}$, there exists a rule $r_i \in \mathcal{C}$ with $body(r_i) \subseteq \mathcal{S} \cup \{q_1, \ldots, q_{i-1}\}$ and $head(r_i) = q_i$. Hence, a subset $\mathcal{S} \subseteq H$ is a solution of the PAP $\mathcal{P}$ iff $\bot \notin \mathcal{S}^+$ and $M \subseteq \mathcal{S}^+$. Our #HORN-ABDUCTION program searches for the number of solutions $\mathcal{S} \subseteq H$ by applying precisely this criterion. The predicate $solve(v, S, i, C^o, RC, \Delta C, RO, j)$, which is at the heart of the #HORN-ABDUCTION program, has the following intended meaning: v denotes a node in the tree decomposition $\mathcal{T}$. S is the projection of a solution $\mathcal{S}$ onto $Hyp(v)$ and C^o is the projection of $\mathcal{S}^+ \setminus \mathcal{S}$ onto $Var(v)$. We consider $\mathcal{S}^+ \setminus \mathcal{S}$ as well as C^o as ordered (which is indicated by the superscript o) w.r.t. some derivation sequence of $\mathcal{S}^+$ from $\mathcal{S}$. The arguments RC, ΔC, and RO are used to check that C^o is indeed the projection of $\mathcal{S}^+ \setminus \mathcal{S}$ onto $Var(v)$. Informally, the arguments RC and ΔC ensure that C^o is not too big, while RO ensures that C^o is not too small. These tasks are accomplished as follows: RC contains those rules in v which are used in the above derivation sequence. Furthermore, the set ΔC contains those variables of C^o, for which we have already found the corresponding derivation rule. Of course, in the bottom-up traversal of the tree decomposition, every element of C^o ultimately has to end up in ΔC. On the other hand, RO contains those rules r in the bag of v which do not constitute a contradiction with the closedness of $\mathcal{S}^+$, i.e., either the head of r is contained in $\mathcal{S}^+$ anyway or we have already encountered in $Var(\mathcal{T}_v)$ a variable in $body(r)$ which is not contained in $\mathcal{S}^+$. The last argument j is used to count the number of different solutions.

In the program, we again use $\cup$ and $\uplus$ to denote ordinary union resp. disjoint union. By $C^o \uplus \{x\}$, we mean that x is arbitrarily "inserted" into C^o, leaving the order of the

Program #HORN-ABDUCTION

/* leaf node. */
$solve(v, S, 0, C^o, RC, \Delta C, RO_1 \cup RO_2, 1) \leftarrow leaf(v)$, $bag(v, X, R)$, $S \cap C^o = \emptyset$, $C^o \subseteq X$, $RC \subseteq R$, $svar(v, S)$, $explains(v, S \cup C^o)$, $consistent(RC, S, C^o, X)$, $derived(\Delta C, C^o, RC)$, $outside(RO_1, R, X \setminus (S \cup C^o))$, $inside(RO_2, R, S \cup C^o)$.

/* variable removal node. */
$aux(v, S, i, 0, C^o, RC, \Delta C, RO, j) \leftarrow bag(v, X, R)$, $child_1(v_1, v)$, $bag(v_1, X \uplus \{x\}, R)$, $solve(v_1, S \uplus \{x\}, i, C^o, RC, \Delta C, RO, j)$.
$aux(v, S, i, 1, C^o, RC, \Delta C, RO, j) \leftarrow bag(v, X, R)$, $child_1(v_1, v)$, $bag(v_1, X \uplus \{x\}, R)$, $solve(v_1, S, i, C^o \uplus \{x\}, RC, \Delta C \uplus \{x\}, RO, j)$.
$aux(v, S, i, 1, C^o, RC, \Delta C, RO, j) \leftarrow bag(v, X, R)$, $child_1(v_1, v)$, $bag(v_1, X \uplus \{x\}, R)$, $solve(v_1, S, i, C^o, RC, \Delta C, RO, j)$, $x \notin S$, $x \notin C^o$.

/* rule removal node. */
$solve(v, S, i, C^o, RC, \Delta C, RO, j) \leftarrow bag(v, X, R)$, $child_1(v_1, v)$, $bag(v_1, X, R \uplus \{r\})$, $solve(v_1, S, i, C^o, RC \uplus \{r\}, \Delta C, RO \uplus \{r\}, j)$.
$solve(v, S, i, C^o, RC, \Delta C, RO, j) \leftarrow bag(v, X, R)$, $child_1(v_1, v)$, $bag(v_1, X, R \uplus \{r\})$, $solve(v_1, S, i, C^o, RC, \Delta C, RO \uplus \{r\}, j)$.

/* variable introduction node. */
$solve(v, S \uplus \{x\}, i, C^o, RC, \Delta C, RO, j) \leftarrow bag(v, X \uplus \{x\}, R)$, $child_1(v_1, v)$, $bag(v_1, X, R)$, $solve(v_1, S, i, C^o, RC, \Delta C, RO, j)$, $hyp(x)$.
$solve(v, S, i, C^o \uplus \{x\}, RC, \Delta C, RO, j) \leftarrow bag(v, X \uplus \{x\}, R)$, $child_1(v_1, v)$, $bag(v_1, X, R)$, $solve(v_1, S, i, C^o, RC, \Delta C, RO, j)$, $consistent(RC, S, C^o \uplus \{x\}, X \uplus \{x\})$.
$solve(v, S, i, C^o, RC, \Delta C, RO_1 \cup RO_2, j) \leftarrow bag(v, X \uplus \{x\}, R)$, $child_1(v_1, v)$, $bag(v_1, X, R)$, $solve(v_1, S, i, C^o, RC, \Delta C, RO_1, j)$, not $man(x)$, $outside(RO_2, R, \{x\})$, $consistent(RC, S, C^o, X \uplus \{x\})$.

/* rule introduction node. */
$solve(v, S, i, C^o, RC \uplus \{r\}, \Delta C \uplus \{x\}, RO \uplus \{r\}, j) \leftarrow bag(v, X, R \uplus \{r\})$, $child_1(v_1, v)$, $bag(v_1, X, R)$, $solve(v_1, S, i, C^o, RC, \Delta C, RO, j)$, $consistent(RC \uplus \{r\}, S, C^o, X)$, $pos(x, r)$, $x \notin \Delta C$, $x \neq \bot$.
$solve(v, S, i, C^o, RC, \Delta C, RO_1 \cup RO_2 \cup RO_3, j) \leftarrow bag(v, X, R \uplus \{r\})$, $child_1(v_1, v)$, $bag(v_1, X, R)$, $solve(v_1, S, i, C^o, RC, \Delta C, RO_1, j)$, $outside(RO_2, R \uplus \{r\}, X \setminus (S \uplus C^o))$, $inside(RO_3, R \uplus \{r\}, S \uplus C^o)$.

/* branch node. */
$aux(v, S, i_1, i_2, C^o, RC, \Delta C_1 \uplus \Delta C_2, RO_1 \uplus RO_2, j_1 * j_2) \leftarrow bag(v, X, R)$, $child_1(v_1, v)$, $bag(v_1, X, R)$, $solve(v_1, S, i_1, C^o, RC, \Delta C_1, RO_1, j_1)$, $child_2(v_2, v)$, $bag(v_2, X, R)$, $solve(v_2, S, i_2, C^o, RC, \Delta C_2, RO_2, j_2)$, $derived(\Delta C, C^o, RC)$, $\Delta C_1 \cap \Delta C_2 = \Delta C$.

/* variable removal and branch node: $aux \Rightarrow solve$ */
$solve(v, S, i, C^o, RC, \Delta C, RO, j) \leftarrow aux(v, S, i_1, i_2, C^o, RC, \Delta C, RO, j')$, $reduce(v, S, i, i_1, i_2, j)$.

/* result (at the root node). */
$count(\text{SUM}(j)) \leftarrow root(v)$, $bag(v, X, R)$, $solve(v, S, i, C^o, RC, \Delta C, RO, j)$, $C^o = \Delta C$, $RO = R$, not $unsuccess(S, i, C^o)$.
$unsuccess(S, i, C_1^o) \leftarrow root(v)$, $bag(v, X, R)$, $solve(v, S, i, C_2^o, RC, \Delta C, RO, j)$, $C_2^o = \Delta C$, $RO = R$, $C_2^o < C_1^o$.

Fig. 5. #HORN-ABDUCTION program

remaining elements unchanged. Analogously to the #CIRCUMSCRIPTION program, we need an index i in order to distinguish between different derivation sequences leading to different orderings on the elements in $\mathcal{S}^+ \setminus \mathcal{S}$. Moreover, we need an *aux*-predicate maintaining pairs of indices in case of variable removable and branch nodes. Moreover, we also need a *reduce*-predicate to contract *aux*-facts $aux(v, S, i_1, i_2, \dots)$ and $aux(v, S, i'_1, i'_2, \dots)$ for partial solutions which are indistinguishable by the *aux*-facts in the lfp. The actual counting and summation is again done in the *reduce*-predicate.

Formally, the correctness of the #HORN-ABDUCTION program can be shown via the Property C defined below. Let $Hyp(v)$, $Man(v)$, $Hyp(\mathcal{T}_v)$, and $Man(\mathcal{T}_v)$ denote the restriction of H and M to the variables in the bag of v or in any bag in $\mathcal{T}_v$, respectively. For arbitrary values of $v, S, C^o, RC, \Delta C$, and RO, we define the following set of extensions $\overline{S}$ of S to $Hyp(\mathcal{T}_v)$:

$Sol(v, S, C^o, RC, \Delta C, RO) = \{\ \overline{S} \mid S \subseteq \overline{S} \subseteq Hyp(\mathcal{T}_v)$ and $\exists \overline{C^o}\, \exists \overline{RC}$ with $C^o \subseteq \overline{C^o} \subseteq Var(\mathcal{T}_v)$ and $RC \subseteq \overline{RC} \subseteq Cl(\mathcal{T}_v)$, s.t.

1. $\overline{S} \cap \overline{C^o} = \emptyset$, $\bot \notin \overline{C^o}$, and $Man(\mathcal{T}_v) \subseteq \overline{S} \cup \overline{C^o}$.
2. $\forall r \in \overline{RC}$, $head(r) \in \overline{C^o}$ and $\forall p \in body(r) \cap Var(\mathcal{T}_v)$: either $p \in \overline{S}$ or $p \in \overline{C^o}$ with $p < head(r)$.
3. $RO = \{r \in Cl(v) \mid body(r) \cap Var(\mathcal{T}_v) \not\subseteq \overline{S} \cup \overline{C^o}\} \cup \{r \in Cl(v) \mid head(r) \in S \cup C^o\}$ and $\forall r \in Cl(\mathcal{T}_v) \setminus Cl(v)$, if $head(r) \notin \overline{S} \cup \overline{C^o}$ then $body(r) \not\subseteq \overline{S} \cup \overline{C^o}$.
4. $\Delta C = \{p \in C^o \mid r \in \overline{RC}, head(r) = p\}$ and $\forall p \in \overline{C^o} \setminus C^o$, $\exists r \in \overline{RC}$ with $head(r) = p$. $\}$

Then, occurrences of the ground facts $solve(v, S, i, C^o, RC, \Delta C, RO, j)$ in the lfp of #HORN-ABDUCTION are determined as follows:

Property C. If $Sol(v, S, C^o, RC, \Delta C, RO) = \emptyset$ then no atom $solve(v, S, \_, C^o, RC, \Delta C, RO, \_)$ is in the lfp of #HORN-ABDUCTION. On the other hand, if $Sol(v, S, C^o, RC, \Delta C, RO) \neq \emptyset$ then the following conditions are fulfilled:

(a) A fact $solve(v, S, \_, C^o, RC, \Delta C, RO, j)$ is in the lfp of #HORN-ABDUCTION iff $|Sol(v, S, C^o, RC, \Delta C, RO)| = j$.

(b) For any further tuple of values $(C^o_1, RC_1, \Delta C_1, RO_1)$ we have $Sol(v, S, C^o, RC, \Delta C, RO) = Sol(v, S, C^o_1, RC_1, \Delta C_1, RO_1)$ iff there exists an index i and a value j, s.t. there are facts $solve(v, S, i, C^o, RC, \Delta C, RO, j)$ and $solve(v, S, i, C^o_1, RC_1, \Delta C_1, RO_1, j)$ in the lfp of #HORN-ABDUCTION.

The other predicates have the following intended meaning: $svar(v, S)$ is used to select sets of hypotheses. It is true for every subset $S \subseteq Hyp(v)$. A fact $explains(v, X)$ is in the lfp iff $Man(v) \subseteq X$. These two predicates are only used to ease the notation at the leaf nodes of $\mathcal{T}$. The remaining predicates *consistent*, *outside*, *inside*, and *derived* take care of the conditions 2 – 4 of the definition of $Sol(v, S, C^o, RC, \Delta C, RO)$ in the following way: A fact $consistent(RC, S, C^o, X)$ is in the lfp iff $\forall r \in RC$ we have $head(r) \in C^o$ and $\forall p \in body(r) \cap X$ it holds that either $p \in S$ or $p \in C^o$ with $p < head(r)$, i.e. the rules in RC are only used to derive greater variables from smaller ones (plus variables from S), cf. condition 2 above. A fact $outside(RO, R, X)$ is in the lfp iff $RO = \{r \in R \mid body(r) \cap X \neq \emptyset\}$. Hence, for $X \subseteq V \setminus \mathcal{S}^+$, the rules in RO do not constitute a contradiction with the closedness of $\mathcal{S}^+$ because their bodies have a variable of X (and, therefore, outside $\mathcal{S}^+$) in their body. A fact $inside(RO, R, X)$ is in the lfp iff $RO = \{r \in R \mid head(r) \in X\}$. Hence, for $X \subseteq \mathcal{S}^+$, the rules in RO do not constitute a contradiction with the closedness of $\mathcal{S}^+$ because their head is inside

this set. A fact $derived(\Delta C, C^o, RC)$ means that ΔC contains those variables of C^o for which RC already contains the rule which is used in the last step of the derivation, i.e., $\Delta C = \{q \in C^o \mid r \in RC, q = head(r)\}$. Analogously to Theorems 2 and 3, the #HORN-ABDUCTION-program in Figure 5 has the following properties:

Theorem 4. *Let $\mathcal{P} = \langle V, H, M, \mathcal{C}\rangle$ be an instance of* #HORN-ABDUCTION*, encoded by a τ_{td}-structure $\mathcal{A}_{td}$. Then, $count(j)$ with $j \geq 1$ is in the lfp of the* #HORN-ABDUCTION*-program evaluated on $\mathcal{A}_{td}$ iff the PAP $\mathcal{P}$ is solvable and has j solutions. Moreover, both the construction of the τ_{td}-structure $\mathcal{A}_{td}$ and the evaluation of the program take time $\mathcal{O}(f(tw(\mathcal{P})) * \|\mathcal{P}\|)$ for some function f, if we assume constant runtime for the arithmetic operations.*

6 Experimental Evaluation

A practical evaluation of the monadic datalog approach presented in earlier work [7] is still missing. So far, datalog programs (like the ones established in [7,8]) only served as a "specification" for an implementation in C++, rather than being used as a method of its own for solving the problem. However, using the datalog approach directly would be very appealing, for instance, for rapid prototyping. Below we report on some first lessons learned when experimenting with implementations of the #SAT program.

When evaluating the #SAT-program on a datalog engine, several obstacles have to be overcome: First, encodings for the non-standard datalog operations, especially those for set arithmetic, are non-trivial and must be done very carefully (avoiding the introduction of cycles, etc.). A recent extension of the DLV-system [11], which is called DLV-Complex (see http://www.mat.unical.it/dlv-complex), provides special built-in predicates for set arithmetic. Our experiments showed that such built-ins normally lead to a better performance than a direct realization of the #SAT-program in "pure" datalog. Another interesting observation was that the DLV-system did not recognize that the $solve()$-predicate can be evaluated without any cycles by a bottom-up traversal of the tree-decomposition. We therefore relaxed the separation of the program and the data and generated the programs using predicates $solve_v()$, for each node v in the tree-decomposition instead of having v as an argument in $solve()$ – thus making the acyclicity explicit. This led to a significant speed-up. Note that we could have put more and more computation tasks into the generation of the datalog program. However, to keep the method generic (w.r.t. different problems) we restricted ourselves to exploit only *structural information*, i.e. the shape of the tree decomposition. Further, DLV is handicapped in the way that no values bigger than 10^{10} can be processed.

We carried out experiments with two implementations of our #SAT program: one executing the datalog program directly on DLV-complex (compiling the tree structure into the program as discussed above) and one using a general-purpose, Turing complete programming language (in contrast to [7,8], we used Haskell rather than C++, because we found it more convenient). Table 1 shows a glimpse of our results for various values of the treewidth (tw), number of variables (# vars), clauses (# clauses) and nodes in the tree decomposition (# nodes). The experiments were done on a recent Core2Duo processor with 2GB of RAM and two cores at 1.86 GHz. The time was measured with the Unix tool "time". DLV was called with the default optimization parameters. Haskell was compiled with increased optimization levels. Comparing a compiled program with an interpreted program might be "unfair", but the Haskell program does not need to be

Table 1. Processing Time in sec. for #SAT.

tw	# vars	# clauses	# nodes	# models	Haskell	datalog
3	75	25	220	2.1E13	0.00	5.67
3	150	50	439	2.2E25	0.00	22.22
3	300	100	949	4.6E54	0.00	177.90
4	75	25	214	9.8E11	0.00	6.07
4	150	50	453	9.0E28	0.00	22.72
4	300	100	950	2.6E52	0.01	233.24
5	300	100	913	2.3E51	0.01	166.72
6	300	100	981	1.7E53	0.02	141.20
7	300	100	979	3.6E52	0.04	259.97
10	309	103	1044	5.1E48	4.12	2841.10

recompiled when the tree changes whereas the DLV program has to be generated for each instance.

In theory, our #SAT algorithm specified in terms of a datalog program is fixed-parameter linear whenever the program is evaluated in an "optimal" way. This is what our Haskell implementation does. For the time being, it is unclear how the design of the datalog program (or the underlying datalog engine) has to be changed such that the datalog engine yields similar results. This is subject of ongoing research. Nevertheless the datalog approach scales reasonably for instances of medium size. Therefore, already now, datalog engines can be employed as tools for rapid prototyping and to verify specifications, which are planned to be realized by a program in another language.

7 Conclusion

We have shown that the monadic datalog approach of [7] can be extended to counting problems defined via MSO. It should be noted that – as opposed to [13,14] – our ultimate goal is not an efficient algorithm for the #SAT problem. Instead we are aiming at a general-purpose method which allows us to systematically turn theoretical tractability results based on Courcelle's Theorem and generalizations thereof into efficient computations. The experiments with our proof-of-concept implementation demonstrate that our goal is realistic even though there is still a lot of work ahead of us.

Analogously to [13,14], our datalog programs ultimately follow a dynamic programming approach. This is not surprising if we keep the crucial observation underlying Courcelle's Theorem in mind: Consider a structure $\mathcal{A}$ with tree decomposition $\mathcal{T}$ and some node s in $\mathcal{T}$. If a domain element a in some bag above s and an element b below s jointly occur in some tuple in $\mathcal{A}$ then – by the definition of tree decompositions – b also occurs in the bag of s. Hence, the essential properties of the substructure induced by the subtree rooted at s can be described in terms of the elements in the bag of s – without taking the concrete form of the subtree rooted at s into account. Indeed, our #SAT program behaves very similar to the dynamic programming algorithm in [14] for the incidence graph. Nevertheless, we find the declarative style of datalog appealing and it has proved convenient in tackling not only #P problems but also the #NP-problem #CIRCUMSCRIPTION. Moreover, the use of datalog allows us to take advantage of all future improvements of datalog engines, which is a very active research area [17].

As future work in this area, we are planning to prove a general expressivity result as to how monadic datalog has to be extended in order to be applicable to any MSO-based counting problem over structures with bounded treewidth. Moreover, we also want to integrate further extensions of Courcelle's Theorem (like sum, minimum, and maximum, which are studied in [3]) into the monadic datalog approach of [7]. As far as our implementation on top of DLV is concerned, we have already identified some directions of future work in Section 6. Note that we have so far used DLV only as a "black box" by converting a #SAT problem instance plus the extended datalog program for #SAT into the DLV syntax. Integrating some of the extensions into the datalog system itself (e.g., an efficient implementation of the *reduce*-predicate in Figures 4 and 5 via hash tables) would clearly help to improve the performance.

References

1. Courcelle, B.: Graph Rewriting: An Algebraic and Logic Approach. In: Handbook of Theoretical Computer Science, vol. B, pp. 193–242. Elsevier Science Publishers, Amsterdam (1990)
2. Gottlob, G., Pichler, R., Wei, F.: Bounded treewidth as a key to tractability of knowledge representation and reasoning. In: Proc. AAAI 2006, pp. 250–256 (2006)
3. Arnborg, S., Lagergren, J., Seese, D.: Easy Problems for Tree-Decomposable Graphs. J. Algorithms 12, 308–340 (1991)
4. Flum, J., Frick, M., Grohe, M.: Query evaluation via tree-decompositions. J. ACM 49, 716–752 (2002)
5. Frick, M., Grohe, M.: The complexity of first-order and monadic second-order logic revisited. In: Proc. LICS 2002, pp. 215–224 (2002)
6. Grohe, M.: Descriptive and Parameterized Complexity. In: Flum, J., Rodríguez-Artalejo, M. (eds.) CSL 1999. LNCS, vol. 1683, pp. 14–31. Springer, Heidelberg (1999)
7. Gottlob, G., Pichler, R., Wei, F.: Monadic Datalog over Finite Structures with Bounded Treewidth. In: Proc. PODS 2007, pp. 165–174 (2007)
8. Gottlob, G., Pichler, R., Wei, F.: Abduction with bounded treewidth: From theoretical tractability to practically efficient computation. In: Proc. AAAI 2008, pp. 1541–1546 (2008)
9. Durand, A., Hermann, M., Kolaitis, P.G.: Subtractive reductions and complete problems for counting complexity classes. Theor. Comput. Sci. 340, 496–513 (2005)
10. Hermann, M., Pichler, R.: Counting complexity of propositional abduction. In: Proc. IJCAI 2007, pp. 417–422 (2007)
11. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM Trans. Comput. Log. 7, 499–562 (2006)
12. Courcelle, B., Makowsky, J.A., Rotics, U.: On the fixed parameter complexity of graph enumeration problems definable in monadic second-order logic. Discrete Applied Mathematics 108, 23–52 (2001)
13. Fischer, E., Makowsky, J.A., Ravve, E.V.: Counting truth assignments of formulas of bounded tree-width or clique-width. Discrete Applied Mathematics 156, 511–529 (2008)
14. Samer, M., Szeider, S.: Algorithms for propositional model counting. In: Dershowitz, N., Voronkov, A. (eds.) LPAR 2007. LNCS, vol. 4790, pp. 484–498. Springer, Heidelberg (2007)
15. Bodlaender, H.L.: A Linear-Time Algorithm for Finding Tree-Decompositions of Small Treewidth. SIAM J. Comput. 25, 1305–1317 (1996)
16. Jakl, M., Pichler, R., Rümmele, S., Woltran, S.: Fast counting with bounded treewidth. Technical Report DBAI-TR-2008-61, Technische Universität Wien (2008)
17. Gebser, M., Liu, L., Namasivayam, G., Neumann, A., Schaub, T., Truszczyński, M.: The first answer set programming system competition. In: Baral, C., Brewka, G., Schlipf, J. (eds.) LPNMR 2007. LNCS, vol. 4483, pp. 3–17. Springer, Heidelberg (2007)

Cut Elimination for First Order Gödel Logic by Hyperclause Resolution

Matthias Baaz*, Agata Ciabattoni**, and Christian G. Fermüller***

Technische Universität Wien, Austria

Abstract. Efficient, automated elimination of cuts is a prerequisite for proof analysis. The method CERES, based on Skolemization and resolution has been successfully developed for classical logic for this purpose. We generalize this method to Gödel logic, an important intermediate logic, which is also one of the main formalizations of fuzzy logic.

1 Introduction

In recent years an efficient method for the automated elimination of cuts from classical first order sequent proofs has been developed [7,9]. This method, called CERES[1] is based on the resolution calculus and has been successfully employed for the in depth analysis of proofs in number theory (e.g., [5]). It is moreover also of theoretical interest due to its global nature and other essential differences, compared to the traditional, local Gentzen- and Schütte-Tait-style cut elimination methods [18,20]. Of course, effective cut elimination is not only useful for classical logic. It is a precondition for non-trivial proof analysis in any logic. In [8] Baaz and Leitsch have extended CERES to a wide class of finite-valued logics. Considering the intended applications, intuitionistic logic and intermediate logics, i.e., logics over the standard language that are stronger than intuitionistic logic, but weaker than classical logic, are even more important targets for similar extensions. However, there are a number formidable obstacles to a straightforward generalization of CERES to this realm of logics:

- It is unclear whether and how classical resolution can be generalized, for the intended purpose, to intermediate logics.
- Gentzen's sequent format is too restrictive to obtain appropriate analytic calculi for many important intermediate logics.
- Skolemization, or rather the inverse de-Skolemization of proofs — an essential prerequisite for CERES — is not possible in general.

Here we single out a prominent intermediate logic, namely Gödel logic **G** (also called Dummett's **LC** or Gödel-Dummett logic), which is also one of the main formalizations of fuzzy logic (see, e.g., [13]) and therefore sometimes called intuitionistic fuzzy

* Partially supported by FWF (Austrian Science Foundation) P19875.
** Partially supported by FWF Project P18731.
*** Partially supported by ESF/FWF Project I143-G15.

[1] CERES stands for Cut Elimination by RESolution.

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 451–466, 2008.

logic [22]. We show that essential features of CERES can be adapted to the calculus HG [1,11] for **G** that uses *hypersequents*, a generalization of Gentzen's sequents to multisets of sequents. This adaption is far from trivial and, among other novel features, entails a new concept of 'resolution': hyperclause resolution, which combines most general unification and cuts on atomic hypersequents. It also provides clues to a better understanding of resolution based cut elimination for sequent and hypersequent calculi, in general.

Due to the incorrectness of general de-Skolemization we will deal with HG-proofs with (arbitrary cut-formulas, but) end-hypersequents that contain either only weak quantifier occurrences or only prenex formulas. For the latter case we show that the corresponding class of proofs admits de-Skolemization.

Our results can also be seen as a first step towards automatizing cut elimination and proof analysis for intuitionistic and other intermediate logics.

2 First Order Gödel Logic and Hypersequents

First-order Gödel logic **G** is one of the most important intermediate logics. It can be characterized semantically by the class of all rooted linearly ordered Kripke models with constant domains, see e.g. [12]. Syntactically, **G** arises from intuitionistic logic by adding the axiom of linearity $(A \supset B) \vee (B \supset A)$ and the quantifier shifting axiom $\forall x(A(x) \vee C) \supset [(\forall x A(x)) \vee C]$, where the x does not occur free in C.

The importance of the logic is also indicated by the fact that it can alternatively be seen as a fuzzy logic, i.e., as an infinite-valued logic with the real unit interval $[0,1]$ as set of truth values [22,13]; but also as a temporal logic [10].

Hypersequent calculi [2] are simple and natural generalizations of Gentzen's sequent calculi. In our context, a hypersequent is a multiset of single-conclusioned ('intuitionistic') sequents, called *components*, written as

$$\Gamma_1 \Rightarrow \Delta_1 \mid \cdots \mid \Gamma_n \Rightarrow \Delta_n$$

where, for $i \in \{1,\ldots,n\}$, Γ_i is a multiset of formulas, and Δ_i is either empty or a single formula. The intended interpretation of the symbol '|' is disjunction at the meta-level.

A hypersequent calculus for propositional Gödel logic has been introduced by Avron [1,2] and extended to first-order in [11]. The logical rules and internal structural rules of this calculus are essentially the same as those in Gentzen's sequent calculus LJ for intuitionistic logic; the only difference being the presence of contexts $\mathscr{H}$ representing (possibly empty) *side hypersequents*. In addition we have *external* contraction and weakening, and the so-called *communication rule*. We present an equivalent version HG of the calculi in [1,11] with multiplicative logical rules (see, e.g., [21] for this terminology).

Axioms: $\bot \Rightarrow$, $\quad A \Rightarrow A$, for *atomic*[2] formulas A.

In the following rules, Δ is either empty or a single formula.

Internal structural rules:

$$\frac{\mathscr{H} \mid \Gamma \Rightarrow \Delta}{\mathscr{H} \mid A, \Gamma \Rightarrow \Delta}\ (iw\text{-}l) \qquad \frac{\mathscr{H} \mid \Gamma \Rightarrow}{\mathscr{H} \mid \Gamma \Rightarrow A}\ (iw\text{-}r) \qquad \frac{\mathscr{H} \mid A, A, \Gamma \Rightarrow \Delta}{\mathscr{H} \mid A, \Gamma \Rightarrow \Delta}\ (ic\text{-}l)$$

[2] The restriction to atomic axioms is useful, but does not imply any loss of generality.

External structural rules:

$$\frac{\mathscr{H}}{\mathscr{H} \mid \Gamma \Rightarrow \Delta}\ (ew) \qquad \frac{\mathscr{H} \mid \Gamma \Rightarrow \Delta \mid \Gamma \Rightarrow \Delta}{\mathscr{H} \mid \Gamma \Rightarrow \Delta}\ (ec)$$

Logical rules:

$$\frac{\mathscr{H} \mid A_1, \Gamma_1 \Rightarrow \Delta \qquad \mathscr{H}' \mid A_2, \Gamma_2 \Rightarrow \Delta}{\mathscr{H} \mid \mathscr{H}' \mid A_1 \vee A_2, \Gamma_1, \Gamma_2 \Rightarrow \Delta}\ (\vee\text{-}l) \qquad \frac{\mathscr{H} \mid \Gamma \Rightarrow A_i}{\mathscr{H} \mid \Gamma \Rightarrow A_1 \vee A_2}\ (\vee_i\text{-}r)_{i\in\{1,2\}}$$

$$\frac{\mathscr{H} \mid A_i, \Gamma \Rightarrow \Delta}{\mathscr{H} \mid A_1 \wedge A_2, \Gamma \Rightarrow \Delta}\ (\wedge_i\text{-}l)_{i\in\{1,2\}} \qquad \frac{\mathscr{H} \mid \Gamma_1 \Rightarrow A \qquad \mathscr{H}' \mid \Gamma_2 \Rightarrow B}{\mathscr{H} \mid \mathscr{H}' \mid \Gamma_1, \Gamma_2 \Rightarrow A \wedge B}\ (\wedge\text{-}r)$$

$$\frac{\mathscr{H} \mid \Gamma_1 \Rightarrow A \qquad \mathscr{H}' \mid B, \Gamma_2 \Rightarrow \Delta}{\mathscr{H} \mid \mathscr{H}' \mid A \supset B, \Gamma_1, \Gamma_2 \Rightarrow \Delta}\ (\supset\text{-}l) \qquad \frac{\mathscr{H} \mid A, \Gamma \Rightarrow B}{\mathscr{H} \mid \Gamma \Rightarrow A \supset B}\ (\supset\text{-}r)$$

In the following quantifier rules t denotes an arbitrary term, and y denotes an *eigenvariable*, i.e., y does not occur in the lower hypersequent:

$$\frac{\mathscr{H} \mid A(t), \Gamma \Rightarrow \Delta}{\mathscr{H} \mid (\forall x)A(x), \Gamma \Rightarrow \Delta}\ (\forall\text{-}l) \qquad \frac{\mathscr{H} \mid \Gamma \Rightarrow A(y)}{\mathscr{H} \mid \Gamma \Rightarrow (\forall x)A(x)}\ (\forall\text{-}r)$$

$$\frac{\mathscr{H} \mid A(y), \Gamma \Rightarrow \Delta}{\mathscr{H} \mid (\exists x)A(x), \Gamma \Rightarrow \Delta}\ (\exists\text{-}l) \qquad \frac{\mathscr{H} \mid \Gamma \Rightarrow A(t)}{\mathscr{H} \mid \Gamma \Rightarrow (\exists x)A(x)}\ (\exists\text{-}r)$$

Like in [21] we call the exhibited formula in the lower hypersequent of each of these rules the *main formula*, and the corresponding subformulas exhibited in the upper hypersequents the *active formulas* of the inference.

The following **communication** rule of HG is specific to logic **G**:

$$\frac{\mathscr{H} \mid \Gamma_1, \Gamma_2 \Rightarrow \Delta_1 \qquad \mathscr{H}' \mid \Gamma_1, \Gamma_2 \Rightarrow \Delta_2}{\mathscr{H} \mid \mathscr{H}' \mid \Gamma_1 \Rightarrow \Delta_1 \mid \Gamma_2 \Rightarrow \Delta_2}\ (com)$$

This version of the communication is equivalent to the one introduced in [1] (see [2]).

Finally we have **cut**, where A is called *cut-formula* of the inference:

$$\frac{\mathscr{H} \mid \Gamma_1 \Rightarrow A \qquad \mathscr{H}' \mid A, \Gamma_2 \Rightarrow \Delta}{\mathscr{H} \mid \mathscr{H}' \mid \Gamma_1, \Gamma_2 \Rightarrow \Delta}\ (cut)$$

If A is atomic we speak of an *atomic cut*.

Remark. Note the absence of negation from our calculus: $\neg A$ is just an abbreviation of $A \supset \bot$. (See, e.g., [21] for similar sytems for intuitionistic logic.)

Communication allows us to derive the following additional 'distribution rule' which we will use in Section 6:

$$\frac{\mathscr{H} \mid \Gamma \Rightarrow A \vee B}{\mathscr{H} \mid \Gamma \Rightarrow A \mid \Gamma \Rightarrow B}\ (distr)$$

A derivation ρ using the rules of HG is viewed as an upward rooted tree. The root of ρ is called its *end-hypersequent*, which we will denote by $\mathscr{H}_\rho$. The leaf nodes are called *initial hypersequents*. A *proof* σ of a hypersequent $\mathscr{H}$ is a derivation with $\mathscr{H}_\sigma = \mathscr{H}$, where all initial hypersequents are axioms.

The *ancestors* of a formula occurrence in a derivation are traced upwards to the initial hypersequents in the obvious way. I.e., active formulas are immediate ancestors of the main formula of an inference. The other formula occurrences in the premises (i.e., upper hypersequents) are immediate ancestors of the corresponding formula occurrences in the lower hypersequent. (This includes also internal and external contraction: here, a formula in the lower hypersequent may have *two* corresponding occurrences, i.e. immediate ancestors, in the premises.) The ancestor relation is the transitive closure of immediate ancestorship.

The sub-hypersequent consisting of all ancestors of cut-formulas of an hypersequent $\mathscr{H}$ in a derivation is called the *cut-relevant part* of $\mathscr{H}$. The complementary sub-hypersequent of $\mathscr{H}$ consisting of all formula occurrences that are not ancestors of cut-formulas is the *cut-irrelevant part* of $\mathscr{H}$. An inference is called *cut-relevant* if its main formula is an ancestor of a cut-formula, and is called *cut-irrelevant* otherwise.

The hypersequent $\Gamma_1 \Rightarrow \Delta_1 \mid \ldots \mid \Gamma_n \Rightarrow \Delta_n$ is called *valid* if its translation $\bigvee_{1\leq i\leq n}\left(\bigwedge_{A\in\Gamma_i} A \supset [\Delta_i]\right)$ is valid in **G**, where $[\Delta_i]$ is $\bot$ if Δ_i is empty, and the indicated implications collapse to Δ_i whenever Γ_i is empty. A set of hypersequents is called *unsatisfiable* if their translations entail $\bot$ in **G**. (Different but equivalent ways of defining validity and entailment in **G** have been indicated at the beginning of this section.)

Theorem 1 ([11,3]). *A hypersequent $\mathscr{H}$ is provable in* HG *without cuts iff $\mathscr{H}$ is valid.*

Remark. It might surprise the reader that we rely on the *cut-free* completeness of HG in a paper dealing with *cut elimination*. However, this just emphasizes the fact that we are interested in a *particular* transformation of proofs with cuts (i.e., 'lemmas') into cut-free proofs, that is adequate for automatization and proof analysis (compare [9,5]).

3 Overview of hyperCERES

Before presenting the details of our transformation of appropriate HG-proofs into cut-free proofs, which we call *hyperCERES*, we will assist the orientation of the reader and describe the overall procedure on a more abstract level using keywords that will be explained in the following sections.[3]

The end-hypersequent $\mathscr{H}_\sigma$ of the HG-proof σ that forms the input of hyperCERES can be of two forms: either it contains only weak quantifier occurrences or it consists of prenex formulas only.[4] In the latter case we have to Skolemize the proof first (step 1) and de-Skolemize it after cut elimination (step 7):

1. if necessary, construct a *Skolemized form* $\hat{\sigma}$ of σ, otherwise $\hat{\sigma} = \sigma$ (Section 4)
2. compute a *characteristic set of pairs* $\{\langle R_1(\hat{\sigma}), D_1\rangle, \ldots \langle R_n(\hat{\sigma}), D_n\rangle\}$, where $\Sigma_d(\hat{\sigma}) = \{D_1, \ldots, D_n\}$ is the *characteristic set of d-hyperclauses* — coding the cut formulas of $\hat{\sigma}$ — and each *reduced proof* $R_i(\hat{\sigma})$ is a cut-free proof of a cut-irrelevant sub-hypersequent of $\mathscr{H}_{\hat{\sigma}}$ augmented by D_i (Section 5)

[3] Due to space constraints we have to refer the reader to [9] for a presentation of *CERES*.

[4] While in classical logic all formulas can be translated into equivalent prenex formulas, this does not hold for **G**.

3. translate $\Sigma_d(\hat{\sigma})$ into an equivalent set of hyperclauses $\Sigma(\hat{\sigma})$ and construct a *(hyperclause) resolution refutation* γ of $\Sigma(\hat{\sigma})$ (Section 6)
4. compute a *ground instantiation* γ' of γ using a ground substitution θ (Section 6)
5. apply θ to the reduced proofs $R_1(\hat{\sigma}), \ldots, R_n(\hat{\sigma})$, and assemble them into a single proof $\gamma'[\hat{\sigma}]$ using the atomic cuts and contractions that come from γ (Section 7)
6. eliminate the atomic cuts in $\gamma'[\hat{\sigma}]$ in the usual way[5]
7. if necessary, *de-Skolemize* the proof $\gamma'[\hat{\sigma}]$ and apply final contractions and weakenings to obtain a cut-free proof of $\mathscr{H}_\sigma$ (Section 4)

It is well known (see, e.g., [19,17]) that there is no elementary bound on the size of shortest cut-free proofs relative to the size of proofs with cuts of the same end-(hyper)sequent. While the non-elementary upper bound on the complexity of cut elimination obviously also applies to hyperCERES it should be pointed out that the global (hyperclause) resolution based method presented here is considerably faster in general, and never essentially slower, than traditional Gentzen- or Schütte-Tait-style cut elimination procedures [1,3]. Moreover, the reliance on most general unification and atomic cuts, i.e., on resolution for the computational kernel of the procedure implies that hyperCERES is a potentially essential ingredient of (semi-)automated analysis of appropriately formalized proofs.

4 Skolemization and De-Skolemization

Like in the original CERES-method [7,9], step 5 of hyperCERES is sound only if end-(hyper)sequents do not contain strong quantifier occurrences. The reason for this is that, in general, the eigenvariable condition might be violated when the reduced proofs (constructed in step 2) are combined with the resolution refutation (constructed in step 3) to replace the original cuts with atomic cuts. Consequently, like in CERES, we first *Skolemize* the proof; i.e., we replace all strong quantifier occurrences with appropriate Skolem terms. (Obviously this is necessary only if there are strong quantifier occurrences at all.) While this transformation is always sound (in fact also for LJ-proofs), the inverse *de-Skolemization*, i.e., the re-introduction of strong quantifier occurrences according to the information coded in the Skolem terms, is unsound in general.[6] However, as we will show below, de-Skolemization is possible for HG-proofs of *prenex* hypersequents (step 7).

By a *prenex hypersequent* we mean a hypersequent in which all formulas are in prenex form, i.e., all formulas begin with a (possibly empty) *prefix* of quantifier occurrences, followed by a quantifier-free formula. If $\Gamma \Rightarrow \Delta$ is a component of a prenex hypersequent, then all existential quantifiers occurring in Γ and all universal quantifiers occurring in Δ are called *strong*. The other quantifier occurrences are called *weak*.

The *Skolemization* $\mathscr{H}^S$ of a prenex hypersequent $\mathscr{H}$ is obtained as follows. In every component $\Gamma \Rightarrow \Delta$ of $\mathscr{H}$, delete each strong quantifier occurrence $\mathsf{Q}x$ and replace all

[5] As is known, atomic cuts in HG-proofs can be moved upwards to the axioms, where they become redundant (see, e.g., [3,1]).

[6] E.g. $\forall x(A(x) \vee B) \Rightarrow A(c) \vee B$ is provable in LJ while its de-Skolemized version $\forall x(A(x) \vee B) \Rightarrow \forall x A(x) \vee B$ is not.

corresponding occurrences of x by the *Skolem term* $f(\overline{y})$, where f is a new function symbol and $\overline{y}$ are the variables of the weak quantifier occurrences in the scope of which $\mathsf{Q}x$ occurs. (If $\mathsf{Q}x$ is not the scope of any weak quantifier then f is a constant symbol.)

Given an HG-proof σ of $\mathscr{H}$ its *Skolemization* $\hat{\sigma}$ is constructed in stages:

1. Replace the end-hypersequent $\mathscr{H}$ of σ by $\mathscr{H}^S$. Recall that this means that every occurrence of a strongly quantified variable x in $\mathscr{H}$ is replaced by a corresponding Skolem term $f(\overline{y})$.
2. Trace the indicated occurrences of x and of the eigenvariable y corresponding to its introduction throughout σ and replace all these occurrences by $f(\overline{y})$, too.
3. Delete the (now) spurious strong quantifiers and remove the corresponding inferences that introduce these quantifiers in σ.
4. For any inference in σ introducing a weakly quantified variable y by replacing $A(t)$ with $\mathsf{Q}yA(y)$, replace all corresponding occurrences of y in Skolem terms $f(\overline{y})$ by t.

It is straightforward to check that $\hat{\sigma}$ is an HG-proof of $\mathscr{H}^S$. (Note that strong quantifier occurrences in ancestors of cut formulas remain untouched by our Skolemization.)

It is shown in [4] that prenex formulas of **G** allow for de-Skolemization. We generalize this result to *proofs* of prenex hypersequents. Our main tool is the following result from [11].

Theorem 2 (Mid-hypersequents). *Any cut-free* HG*-proof* σ *of a prenex hypersequent* $\mathscr{H}$ *can be stepwise transformed into one in which no propositional rule is applied below any application of a quantifier rule.*

We call a hypersequent $\overline{\mathscr{H}_S}$ a *linked Skolem instance* of $\mathscr{H}$ if each formula A in $\overline{\mathscr{H}_S}$ is an instance of a Skolemized formula A^S that occurs in $\mathscr{H}^S$ on the same side (left or right) of a component as A. Moreover we link A to A^S. As we will see in Section 7, we obtain (cut-free proofs of) linked Skolem instances from step 5 (and 6) of hyperCERES.

Theorem 3 (De-Skolemization). *Given a cut-free* HG*-proof* $\hat{\rho}$ *of a linked Skolem instance* $\overline{\mathscr{H}_S}$ *of a prenex hypersequent* $\mathscr{H}$*, we can find a* HG*-proof* ρ *of* $\mathscr{H}$*.*

Proof. We construct ρ in stages as follows:

1. By applying Theorem 2 to $\hat{\rho}$ we obtain a proof ρ' of the following form:

$$\dfrac{\begin{array}{ccccc}\rho_1^p & \cdots & \rho_i^p & \cdots & \rho_n^p \\ \mathscr{G}_1 & & \mathscr{G}_i & & \mathscr{G}_n \\ & \ddots & \rho^{\mathsf{Q}} & \iddots & \end{array}}{\overline{\mathscr{H}_S}}$$

   where the mid-hypersequents $\mathscr{G}_1, \ldots, \mathscr{G}_n$ separate ρ' into a part ρ^{Q} containing only (weak) quantifier introductions and applications of structural rules and parts $\rho_1^p, \ldots, \rho_n^p$ containing only propositional and structural inferences.
2. Applications of the weakening rules, $(iw\text{-}l)$ and (ew), can be shifted upwards to the axioms in the usual manner, while applications of $(iw\text{-}r)$ can be safely deleted by replacing each axiom $\bot \Rightarrow$ in the proof by $\bot \Rightarrow \Delta$ for suitable Δ.
   Consequently, ρ^{Q} does not contain weakenings after this transformation step.

3. Note that — in contrast to LK — Theorem 2 induces many and not just one mid-hypersequents, in general. The reason for this is the possible presence of the *binary* structural rule (com) in ρ^{Q}. To obtain a proof ρ'' with a single mid-hypersequent, we have to move 'communications' upwards in ρ^{Q}; i.e., we have to permute applications of (com) with applications of (ic), (ec), $(\forall\text{-}l)$, and $(\exists\text{-}r)$, respectively. The only non-trivial case is $(\forall\text{-}l)$. Disregarding side-hypersequents, the corresponding transformation consists in replacing

$$\dfrac{\dfrac{\Gamma, P(x), \Sigma \Rightarrow \Delta}{\Gamma, \forall x P(x), \Sigma \Rightarrow \Delta}\,(\forall\text{-}l) \qquad \Gamma, \forall x P(x), \Sigma \Rightarrow \Delta'}{\Gamma, \forall x P(x) \Rightarrow \Delta \mid \Sigma \Rightarrow \Delta'}\,(com)$$

by

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, P(x), \Sigma \Rightarrow \Delta}{\Gamma, P(x), \Sigma, \Gamma, \forall x P(x) \Rightarrow \Delta}\,(iw)^* \quad \dfrac{\Gamma, \forall x P(x), \Sigma \Rightarrow \Delta'}{\Gamma, P(x), \Sigma, \Gamma, \forall x P(x) \Rightarrow \Delta'}\,(iw)^*}{\Gamma, P(x), \Sigma \Rightarrow \Delta' \mid \Gamma, \forall x P(x) \Rightarrow \Delta}\,(com) \qquad \Gamma, P(x), \Sigma \Rightarrow \Delta}{\Gamma, P(x) \Rightarrow \Delta \mid \Sigma \Rightarrow \Delta' \mid \Gamma, \forall x P(x) \Rightarrow \Delta}\,(com)}{\Gamma, \forall x P(x) \Rightarrow \Delta \mid \Sigma \Rightarrow \Delta' \mid \Gamma, \forall x P(x) \Rightarrow \Delta}\,(\forall\text{-}l)}{\Gamma, \forall x P(x) \Rightarrow \Delta \mid \Sigma \Rightarrow \Delta'}\,(ec)$$

4. For the final step we proceed like in [4], where the soundness of re-introducing strong quantifier occurrences for corresponding Skolem terms is shown: we ignore ρ'' and, given $\mathscr{H}$ and the links to its formulas, apply appropriate inferences to the mid-hypersequent as follows.
   (a) Infer all weak quantifier occurrences, which can be introduced at this stage according to the quantifier prefixes in $\mathscr{H}$.
   (b) Apply all possible internal and external contractions.
   (c) Among the strong quantifiers that immediately precede the already introduced quantifiers we pick one linked to an instance of a Skolem term, that is maximal with respect to the subterm ordering. This term is replaced everywhere by the eigenvariable of the corresponding strong quantifier inference.

   These three steps are iterated until the original hypersequent $\mathscr{H}$ is restored. □

5 Characteristic Hyperclauses and Reduced Proofs

All information of the original HG-proof σ that goes into the cut-formulas is collected in a set $\Sigma_d(\hat{\sigma})$, consisting of hypersequents whose components only contain atomic formulas on the left hand sides and a (possibly empty) disjunction of atomic formulas, on the right hand side. We will call hypersequents of this latter form *d-hyperclauses*. In the proof of Theorem 4 we will construct *characteristic d-hyperclauses* D_i together with corresponding *reduced proofs* $R_i(\hat{\sigma})$ which combine the cut-irrelevant part of the Skolemized proof $\hat{\sigma}$ with D_i. The pairs $\langle R_i(\hat{\sigma}), D_i \rangle$ provide the information needed to construct corresponding proofs containing only atomic cuts.

To assist concise argumentation we assume that the components of all hypersequents in a proof are labelled with unique sets of identifiers. More precisely, a derivation σ is *labelled* if there is a function from all components of hypersequents occurring in σ into the powerset of a set of *identifiers*, satisfying the following conditions: (We will put the label above the corresponding sequent arrow.)

- All components occurring in initial hypersequents of σ are assigned pairwise different singleton sets of identifiers.
- In all unary inferences the labels are transferred from the upper hypersequent to the lower hypersequent in the obvious way. In external weakening (ew) a fresh singleton set is assigned to the new component in the lower hypersequent. In external contraction (ec), if $\Gamma \overset{M}{\Rightarrow} \Delta$ and $\Gamma \overset{N}{\Rightarrow} \Delta$ are the two contracted components of the upper hypersequent, then $\Gamma \overset{M \cup N}{\Rightarrow} \Delta$ is the corresponding component in the lower hypersequent.
- In all binary logical inferences the labels in the side-hypersequents are transferred in the obvious way, and the label of the component containing the main formula is the union of the labels of the components containing the active formulas.
- In (cut) the labels of the components containing the cut formulas are merged, like above, to obtain the label of the exhibited component of the lower hypersequent.
- In (com) the labels of all components are transferred from the premises to the lower hypersequent simply in the same sequence as exhibited in the statement of the rule.

Let $\mathscr{H}$ and $\mathscr{G}$ denote the labelled hypersequents

$$\Gamma_1 \overset{K_1}{\Rightarrow} \Delta_1 \mid \cdots \mid \Gamma_k \overset{K_k}{\Rightarrow} \Delta_k \mid \mathscr{H}' \quad \text{and} \quad \Gamma_1' \overset{K_1}{\Rightarrow} \Delta_1' \mid \cdots \mid \Gamma_k' \overset{K_k}{\Rightarrow} \Delta_k' \mid \mathscr{G}'$$

respectively, where the labels in $\mathscr{H}'$ and $\mathscr{G}'$ are pairwise different and also different from the labels $K_1, \ldots, K_k$. Then $\mathscr{H} \odot \mathscr{G}$ denotes the *merged* hypersequent

$$\Gamma_1, \Gamma_1' \overset{K_1}{\Rightarrow} \Delta_1 \vee \Delta_1' \mid \cdots \mid \Gamma_k, \Gamma_k' \overset{K_k}{\Rightarrow} \Delta_k \vee \Delta_k' \mid \mathscr{H}' \mid \mathscr{G}'$$

where $\Delta_i \vee \Delta_i'$ is Δ_i if Δ_i' is empty and is Δ_i' if Δ_i is empty (and thus $\Delta_i \vee \Delta_i'$ is empty if both are empty).

Theorem 4. *Given a Skolemized and labelled* HG*-proof* $\hat{\sigma}$ *of* $\mathscr{H}_{\hat{\sigma}}$ *one can construct a* characteristic set of pairs $\{\langle R_1(\hat{\sigma}), D_1\rangle, \ldots \langle R_n(\hat{\sigma}), D_n\rangle\}$*, where, for all* $i \in \{1, \ldots, n\}$*,* D_i *is a labelled d-hyperclause and* $R_i(\hat{\sigma})$ *is a labelled '*(reduced)*' cut-free* HG*-proof with the following properties:*

(1) the end-hypersequent of $R_i(\hat{\sigma})$ *is* $\mathscr{H}'_{\hat{\sigma}} \odot D_i$*, for some sub-hypersequent* $\mathscr{H}'_{\hat{\sigma}}$ *of* $\mathscr{H}_{\hat{\sigma}}$*,*
(2) the characteristic d-hyperclause set $\Sigma_d(\hat{\sigma}) = \{D_1, \ldots, D_n\}$ *is unsatisfiable.*

Proof. To show (1) and (2) we use the following induction hypotheses:

(1') A characteristic set of pairs $\langle R_i(\hat{\sigma}'), D_i'\rangle$ exists for every sub-proof $\hat{\sigma}'$ of $\hat{\sigma}$, where $R_i(\hat{\sigma}')$ proves $\mathscr{H}'_{\hat{\sigma}'} \odot D_i'$ for some sub-hypersequent $\mathscr{H}'_{\hat{\sigma}'}$ of $\mathscr{H}_{\hat{\sigma}'}$ which is cut-irrelevant with respect to the original cuts in $\hat{\sigma}$. Moreover, the right hand sides in $\mathscr{H}'_{\hat{\sigma}'} \odot D_i'$ are formulas in either $\mathscr{H}'_{\hat{\sigma}'}$ or in D_i'.

(2') There is a derivation of the cut-relevant part of $\mathscr{H}_{\hat{\sigma}'}$ from the set $\{D'_1,\ldots,D'_m\}$ of d-hyperclauses constructed for $\hat{\sigma}'$.

Note that (2) follows from (2') as the cut-relevant part of $\mathscr{H}_{\hat{\sigma}}$ is an empty hypersequent by definition. The proof proceeds by induction on the length of $\hat{\sigma}'$.

If $\hat{\sigma}'$ consists just of an axiom $A \overset{M}{\Rightarrow} A$ then there is only one pair $\langle R(\hat{\sigma}'), D\rangle$ in the corresponding characteristic set. $R(\hat{\sigma}')$ is the axiom itself and D is the cut-relevant part of $A \overset{M}{\Rightarrow} A$ (which might be the empty hypersequent). (1') and (2') trivially hold. Axioms of the form $\bot \Rightarrow$ are handled in the same way.

If $\hat{\sigma}'$ is not an axiom we distinguish cases according to the last inference in $\hat{\sigma}'$.

($\vee$-l): $\hat{\sigma}'$ ends with the inference

$$\dfrac{\begin{array}{cc}\vdots\,\hat{\rho} & \vdots\,\hat{\tau}\\ \mathscr{H} \mid A_1, \Gamma_1 \overset{M}{\Rightarrow} \Delta & \mathscr{H}' \mid A_2, \Gamma_2 \overset{N}{\Rightarrow} \Delta\end{array}}{\mathscr{H} \mid \mathscr{H}' \mid A_1 \vee A_2, \Gamma_1, \Gamma_2 \overset{M\cup N}{\Rightarrow} \Delta}\ (\vee\text{-}l)$$

By induction hypothesis (1') there are characteristic sets of pairs $S_1 = \{\langle R_1(\hat{\rho}), E_1\rangle, \ldots, \langle R_m(\hat{\rho}), E_m\rangle\}$ and $S_2 = \{\langle R_1(\hat{\tau}), F_1\rangle, \ldots, \langle R_n(\hat{\tau}), F_n\rangle\}$, where the reduced proofs $R_i(\hat{\rho})$ and $R_j(\hat{\tau})$ end in $\mathscr{H}_{R_i(\hat{\rho})} = \mathscr{G}_i \odot E_i$ and in $\mathscr{H}_{R_i(\hat{\tau})} = \mathscr{G}'_j \odot F_j$, respectively, where $\mathscr{G}_i$ and $\mathscr{G}'_j$ are sub-hypersequents of the cut-irrelevant parts of $\mathscr{H} \mid A_1, \Gamma_1 \overset{M}{\Rightarrow} \Delta$ and $\mathscr{H}' \mid A_2, \Gamma_2 \overset{N}{\Rightarrow} \Delta$, respectively. Moreover, by (2'), there are derivations ρ_C and τ_C of the cut-relevant parts of the just mentioned hypersequents from $\{E_1,\ldots,E_m\}$ and $\{F_1,\ldots,F_n\}$, respectively.

Two cases can occur:

(a) If the inference is *cut-relevant*, then the characteristic set S of pairs corresponding to $\hat{\sigma}'$ is just $S_1 \cup S_2$. Condition (1') trivially remains satisfied. Also (2') is maintained because we obtain a derivation of the cut-relevant part of $\mathscr{H} \mid \mathscr{H}' \mid A_1 \vee A_2, \Gamma_1, \Gamma_2 \overset{M\cup N}{\Rightarrow} \Delta$ by joining ρ_C and τ_C with the indicated application of $(\vee\text{-}l)$.

(b) If the inference is *cut-irrelevant*, then we obtain the set S corresponding to $\hat{\sigma}'$ by

$$S = \{\langle R_{ij}(\hat{\rho} \bowtie_{\vee\text{-}l} \hat{\tau}), E_i \bowtie_{ij} F_j\rangle : 1 \le i \le m, 1 \le j \le n\},$$

where $R_{ij}(\hat{\rho} \bowtie_{\vee\text{-}l} \hat{\tau})$ and $E_i \bowtie_{ij} F_J$ are defined as follows.

1. If A_1 does not occur at the indicated position in $\mathscr{H}_{R_i(\hat{\rho})}$ then $R_{ij}(\hat{\rho} \bowtie_{\vee\text{-}l} \hat{\tau})$ is $R_i(\hat{\rho})$ and $E_i \bowtie_{ij} F_j$ is E_i.
2. If A_2 does not occur at the indicated position in $\mathscr{H}_{R_j(\hat{\tau})}$ then $R_{ij}(\hat{\rho} \bowtie_{\vee\text{-}l} \hat{\tau})$ is $R_j(\hat{\tau})$ and $E_i \bowtie_{ij} F_j$ is F_j.
3. If neither A_1 nor A_2 occur as indicated in the reduced proofs, then $R_{ij}(\hat{\rho} \bowtie_{\vee\text{-}l} \hat{\tau})$ can be non-deterministically chosen to be either $R_i(\hat{\rho})$ or $R_j(\hat{\tau})$ and $E_i \bowtie_{ij} F_j$ is either E_i or F_j, accordingly.
4. If both A_1 and A_2 occur at the indicated positions, then $E_i \bowtie_{ij} F_j$ is $E'_i \odot F'_j$, where E'_i (F'_j) is like E_i (F_j), except for changing the label M (N) to $M \cup N$.
   Note that our labelling mechanism guarantees that the appropriate components are identified in merging hypersequents.
   The corresponding reduced proof $R_{ij}(\hat{\rho} \bowtie_{\vee\text{-}l} \hat{\tau})$ is constructed as follows. Since A_1 and A_2 occur as exhibited in the end-hypersequents $\mathscr{G}_i \odot E_i$ and $\mathscr{G}'_j \odot F_j$ of $R_i(\hat{\rho})$ and $R_j(\hat{\tau})$, respectively, we want to join them by introducing $A_1 \vee A_2$ using $(\vee\text{-}l)$ like in $\hat{\sigma}'$. However, $(\vee\text{-}l)$ is only applicable if the right hand sides of

the two relevant components in the premises are identical. To achieve this, we might first have to apply $(\vee\text{-}r)$ or $(iw\text{-}r)$ to the mentioned end-hypersequents. The resulting new end-hypersequent might still contain different components transferred from E_i and F_j, respectively, that need to be merged with other components. This can be achieved by first applying internal weakenings to make the relevant components identical, and then applying external contraction (ec) to remove redundant copies of identical components.

Note that in all four cases (1') remains satisfied by definition of $R_{ij}(\hat{\rho} \bowtie_{\vee\text{-}l} \hat{\tau})$ and of $E_i \bowtie_{ij} F_j$. For cases 1, 2, and 3 also (2') trivially still holds. To obtain (2') for case 4, we proceed in two steps. First we merge the occurrences of clauses $E_1, \ldots, E_m$ in the derivation ρ_C of the cut-relevant part $\mathscr{H}_c^{\hat{\rho}}$ of $\mathscr{H}_{\hat{\rho}}$ with clauses in $\{F_1, \ldots, F_n\}$ to obtain a derivation $\rho_C(F_i)$ of $\mathscr{H}_c^{\hat{\rho}} \odot F_i$ for each $i \in \{1, \ldots, n\}$. In a second step, each initial hypersequent F_i in the derivation τ_C of the cut-relevant part of $\mathscr{H}_{\hat{\tau}}$ is replaced by $\rho_C(F_i)$. By merging also the inner nodes of τ_C with $\mathscr{H}_c^{\hat{\rho}}$ we arrive at a derivation of the cut-relevant part of $\mathscr{H}_{\hat{\sigma}'}$. (Actually, as the rules of **HG** are multiplicative, redundant copies of identical formulas might arise, that are to be removed by finally applying corresponding contractions.)

$(\wedge_i\text{-}l)$, $(\supset\text{-}r)$, $(\vee\text{-}r)$, $(\forall\text{-}l)$, $(\forall\text{-}r)$, $(\exists\text{-}l)$, $(\exists\text{-}r)$, $(ic\text{-}l)$: If the indicated last (unary) inference is *cut-relevant*, then the characteristic set of pairs remains the same as for the sub-proof ending with the premise of this inference.

If the inference is *cut-irrelevant*, then the hyperclauses $E_1, \ldots, E_m$ of the pairs in characteristic set $\{\langle R_1(\hat{\rho}), E_1\rangle, \ldots \langle R_m(\hat{\rho}), E_m\rangle\}$ for $\hat{\rho}$ remain unchanged. Each reduced proof $R_i(\hat{\rho})$ is augmented by the corresponding inference if its active formula occurs in the end-hypersequent $\mathscr{H}_{R_i(\hat{\rho})}$. If this is not the case then also $R_i(\hat{\rho})$ remains unchanged.

In any of these cases, (1') and (2') clearly remain satisfied.

(ew), $(iw\text{-}l)$, $(iw\text{-}r)$: The characteristic set of pairs remains unchanged and consequently (1') still holds. Also (2') trivially remains valid if the inference is cut-irrelevant. If a cut-relevant formula is introduced by weakening, then the derivation required for (2') is obtained from the induction hypothesis by adding a corresponding application of a weakening rule.

$(\wedge\text{-}r)$, $(\supset\text{-}l)$, (cut), (com): These cases are analogous to the one for $(\vee\text{-}l)$. □

Example 1. Consider the labelled proof σ in Figure 1.

The cut-relevant parts of σ and the names of all corresponding cut-relevant inferences are underlined. The initial pair for the $\{1\}$-labelled axiom is $\langle \rho_1, \overset{\{1\}}{\Rightarrow} Q\rangle$, where ρ_1 is $Q \overset{\{1\}}{\Rightarrow} Q$. Since the succeeding inference $(\vee\text{-}r)$ is unary and cut-relevant, the pair remains unchanged in that step.

For the middle part of the proof let us look at the subproof σ' ending with an application of (com) yielding $Q \overset{\{2\}}{\Rightarrow} \exists y P(y) \mid P(c) \overset{\{3\}}{\Rightarrow} Q$. Since there are no cut-ancestors in the $\{2\}$-labelled axiom, the corresponding d-hyperclause is the empty $\overset{\{2\}}{\Rightarrow}$. This is retained for the right premise of (com). The corresponding reduced derivation consists only of the first inference $(\exists\text{-}r)$ as the succeeding application of $(iw\text{-}l)$ is cut-relevant. For the left premise of the communication we obtain the d-hyperclause $Q \overset{\{3\}}{\Rightarrow}$, which is then merged and 'communicated' with $\overset{\{2\}}{\Rightarrow}$ to obtain for σ' the d-hyperclause $Q \overset{\{2\}}{\Rightarrow} \mid \overset{\{3\}}{\Rightarrow}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{P(c) \overset{\{2\}}{\Rightarrow} P(c)}{P(c) \overset{\{2\}}{\Rightarrow} \exists y P(y)}\,(\exists\text{-}r)}{\underline{Q}, P(c) \overset{\{2\}}{\Rightarrow} \exists y P(y)}\,(iw\text{-}l) \quad \dfrac{\underline{Q} \overset{\{3\}}{\Rightarrow} Q}{\underline{Q}, P(c) \overset{\{3\}}{\Rightarrow} Q}\,(iw\text{-}l)}{\underline{Q} \overset{\{2\}}{\Rightarrow} \exists y P(y) \mid P(c) \overset{\{3\}}{\Rightarrow} Q}\,(com) \quad \dfrac{\dfrac{\underline{P(x)} \overset{\{4\}}{\Rightarrow} P(x)}{\underline{P(x)} \overset{\{4\}}{\Rightarrow} \exists y P(y)}\,(\exists\text{-}r)}{}}{\dfrac{\dfrac{Q \overset{\{1\}}{\Rightarrow} \underline{Q}}{Q \overset{\{1\}}{\Rightarrow} \underline{P(x) \vee Q}}\,(\vee\text{-}r)}{\underline{P(x) \vee Q} \overset{\{2,4\}}{\Rightarrow} \exists y P(y) \mid P(c) \overset{\{3\}}{\Rightarrow} Q}\,(\vee\text{-}l) \quad Q \overset{\{5\}}{\Rightarrow} Q}{\underline{P(x) \vee Q} \overset{\{2,4\}}{\Rightarrow} \exists y P(y) \mid P(c) \vee Q \overset{\{3,5\}}{\Rightarrow} Q}\,(cut)$$

$$Q \overset{\{1,2,4\}}{\Rightarrow} \exists y P(y) \mid P(c) \vee Q \overset{\{3,5\}}{\Rightarrow} Q$$

Fig. 1. Labelled proof σ with underlined cut-relevant part

This forms a pair with the reduced derivation $R(\sigma')$, which, in this case, is identical with σ'.[7]

From the cut-relevant (and therefore underlined) $(\vee\text{-}l)$-inference one obtains an additional pair $\langle \rho_2, P(x) \overset{\{4\}}{\Rightarrow} \rangle$ from its right premise, where ρ_2 is the derivation of $P(x) \overset{\{4\}}{\Rightarrow} \exists y P(y)$ from the axiom.

For the succeeding cut-irrelevant application of $(\vee\text{-}l)$, the pair $\langle \rho_2, P(x) \overset{\{4\}}{\Rightarrow} \rangle$ remains unchanged, as the left disjunct $P(x)$ does not occur at the left side in the end-hypersequent $\overset{\{4\}}{\Rightarrow} \exists y P(y)$ of ρ_2.[8] The reduced proof ρ_3 of the final pair is formed by applying $(\vee\text{-}l)$ as indicated to the end-hypersequent of $R(\sigma')$ and to $Q \overset{\{5\}}{\Rightarrow} Q$ as right and left premises, respectively. The corresponding d-hyperclause arises from merging $Q \overset{\{2\}}{\Rightarrow} \mid \overset{\{3\}}{\Rightarrow}$ and $\overset{\{5\}}{\Rightarrow}$ into $Q \overset{\{2\}}{\Rightarrow} \mid \overset{\{3,5\}}{\Rightarrow}$.

For the final application of cut we have to take the union of the sets of pairs constructed for its two premises. Therefore the characteristic set of pairs for σ is

$$\{\langle \rho_1, \overset{\{1\}}{\Rightarrow} Q \rangle,\ \langle \rho_2, P(x) \overset{\{4\}}{\Rightarrow} \rangle,\ \langle \rho_3, Q \overset{\{2\}}{\Rightarrow} \mid \overset{\{3,5\}}{\Rightarrow} \rangle\}\ .$$

It is easy to check that conditions (1) and (2) of Theorem 4 are satisfied.

6 Hyperclause Resolution

By a *hyperclause* we mean a hypersequent in which only atomic formulas occur. Remember that, from the proof of Theorem 4, we obtain *d-hyperclauses*, which are like hyperclauses, except for allowing *disjunctions* of atomic formulas at the right hand sides of their components. However, using the derivable rule (*distr*) (see Section 2) it is easy to see that an HG-derivation of, e.g., the d-hyperclause

$$A \Rightarrow B \vee C \mid \Rightarrow D \vee E \vee F$$

[7] Note that neither the cut-relevant application of $(iw\text{-}l)$ nor Q appears in the reduced proof corresponding to $Q, P(c) \overset{\{2\}}{\Rightarrow} \exists y P(y)$. Still, the missing Q is added by $(iw\text{-}l)$ in $R(\sigma')$ to make the application of (com) possible.

[8] This is case $(\vee\text{-}l)$/(b)/2 in the proof of Theorem 4.

can be replaced by an HG-derivation of the hyperclause

$$A \Rightarrow B \mid A \Rightarrow C \mid \Rightarrow D \mid \Rightarrow E \mid \Rightarrow F \ .$$

Also the converse holds: using the rules $(\vee_i\text{-}r)$, and (ec) we can derive the mentioned d-hyperclause from the latter hyperclause. Therefore we can refer to hyperclauses instead of d-hyperclauses in the following.

We also want to get rid of occurrences of $\bot$ in hyperclauses. Since $\bot \Rightarrow$ is an axiom, any hyperclause which contains an occurrence of $\bot$ at the left hand side of some component is valid. But such hyperclauses are redundant, as our aim is to construct *refutations* for unsatisfiable sets of hyperclauses. On the other hand, any occurrence of $\bot$ at the right hand side of a component is also redundant and can be deleted. In other words: we can assume without loss of generality that $\bot$ does not occur in hyperclauses. (Note that this does not imply that occurrences of $\bot$ are removed from HG-proofs.)

In direct analogy to classical resolution, the combination of a cut-inference and most general unification is called a *resolution* step. The lower hyperclause in

$$\frac{\mathscr{H} \mid \Gamma_1 \Rightarrow A \quad \mathscr{H}' \mid A', \Gamma_2 \Rightarrow \Delta}{\theta(\mathscr{H} \mid \mathscr{H}' \mid \Gamma_1, \Gamma_2 \Rightarrow \Delta)} \ (res)$$

where θ is the most general unifier of the atoms A and A', is called *resolvent* of the premises, that have to be variable disjoint. If no variables occur, and thus θ is empty, (res) turns into (cut) and we speak of *ground resolution*. The soundness of this inference step is obvious. We show that hyperclause resolution is also refutationally complete. It is convenient to view hyperclauses as sets of atomic sequents. This is equivalent to requiring that external contraction is applied whenever possible. Consequently, there is a unique unsatisfiable hyperclause, namely the *empty hyperclause*. A derivation of the empty hyperclause by resolution from initial hypersequents contained in a set Σ of hyperclauses is called a *resolution refutation* of Σ.

As usual for resolution, we focus on inferences on ground hyperclauses and later transfer completeness to the general level using a corresponding *lifting lemma*.

Theorem 5. *For every unsatisfiable set of ground hyperclauses Ψ there is a ground resolution refutation of Ψ.*

Proof. We proceed by induction on $e(\Psi) = \|\Psi\| - |\Psi|$, where $\|\Psi\|$ is the total number of occurrences of atoms in Ψ, and $|\Psi|$ is the cardinality of Ψ.

If $e(\Psi) \leq 0$ then either Ψ already contains the empty hyperclause, or else Ψ contains exactly one atom per hyperclause. In the latter case, as Ψ is unsatisfiable, there must be hyperclauses $C_1 = (\Rightarrow A)$ and $C_2 = (A \Rightarrow)$ in Ψ. Obviously the empty clause is a ground resolvent of C_1 and C_2.

$e(\Psi) \geq 1$: Ψ must contain a hyperclause C that has more than one atom occurrence. Without loss of generality let $C = (\mathscr{H} \mid \Gamma \Rightarrow A)$, where Γ may be empty. (The case where all atoms in C occur only on the left hand side of sequents is analogous.) Since Ψ is unsatisfiable also the sets $\Psi' = (\Psi - \{C\}) \cup \{\mathscr{H} \mid \Gamma \Rightarrow\}$ and $\Psi'' = (\Psi - \{C\}) \cup \{\Rightarrow A\}$ must be unsatisfiable. Since $e(\Psi') < e(\Psi)$ and $e(\Psi'') < e(\Psi)$ we obtain ground resolution refutations ρ' of Ψ' and ρ'' of Ψ'', respectively. By adding

in ρ' an occurrence of A to the right side of the derived empty hyperclause and likewise to all other hyperclauses in ρ' that are on a branch ending in the initial hyperclause $\mathscr{H} \mid \Gamma \Rightarrow$, we obtain a resolution derivation ρ'_A of $\Rightarrow A$ from Ψ. By replacing each occurrence of $\Rightarrow A$ as initial hyperclauses in ρ'' by a copy of ρ'_A we obtain the required ground resolution refutation of Ψ. □

Remark. Note that our completeness proof does not use any special properties of **G**. Only the polarity between left and right hand side of sequent and the disjunctive interpretation of '|' at the meta-level are used. For any logic $\mathscr{L}$: whenever we can reduce $\mathscr{L}$-validity (or $\mathscr{L}$-unsatisfiability) of a formula F to $\mathscr{L}$-unsatisfiability of a corresponding set of atomic hyperclauses, we may use hyperclause resolution to solve the problem.

To lift Theorem 5 to general hyperclauses, one needs to add (the hypersequent version of) *factorization*:

$$\frac{\mathscr{H} \mid \Gamma \Rightarrow \Delta}{\theta(\mathscr{H} \mid \Gamma' \Rightarrow \Delta)} \ (factor)$$

where θ is the most general unifier (see, e.g., [15]) of some atoms in Γ and where $\theta\Gamma'(\theta)$ is $\theta(\Gamma)$ after removal of copies of unified atoms. The lower hyperclause is called a *factor* of the upper one.

Lemma 1. *Let C'_1 and C'_2 be ground instances of the variable disjoint hyperclauses C_1 and C_2, respectively. For every ground resolvent C' of C'_1 and C'_2 there is a resolvent C of factors of C_1 and C_2.*

The proof of Lemma 1 is exactly as for classical resolution (see, e.g., [15]) and thus is omitted here. Combining Theorem 5 and Lemma 1 we obtain the refutational completeness of general resolution.

Corollary 1. *For every unsatisfiable set of hyperclauses Σ there is a resolution refutation of Σ.*

We will make use of the observation that any general resolution refutation of Σ can be *instantiated* into (essentially) a ground resolution refutation of a set Σ' of instances of hyperclauses in Σ, whereby resolution steps turn into cuts and factorization turns into additional contraction steps. (Note that additional contractions do not essentially change the structure of a ground resolution refutation.)

7 Projection of Hyperclauses into ⊢G-Proofs

Remember that from Theorem 4 (in Section 5) we obtain a characteristic set of pairs $\{\langle R_1(\hat{\sigma}), D_1\rangle, \ldots \langle R_n(\hat{\sigma}), D_n\rangle\}$ for the proof $\hat{\sigma}$ of $\mathscr{H}^S$. As described in Section 6, we can construct a resolution refutation γ of the hyperclause set $\{C_1, \ldots, C_n\}$ corresponding to the d-hyperclauses $\{D_1, \ldots, D_n\}$. (This is step 3 of hyperCERES.) Forming a ground instantiation γ' of γ yields a derivation of the empty hypersequent that consists only of atomic cuts and contractions. (Step 4 of hyperCERES.) Each leaf node of γ' is a ground instance $\theta(C_i)$ of a hyperclause in $\{C_1, \ldots, C_n\}$. From Theorem 4 we also obtain, for

each $i \in \{1,\ldots,n\}$ a cut-free proof $R_i(\hat{\sigma})$ of $\mathscr{G}_i \odot D_i$, where $\mathscr{G}_i$ is a sub-hypersequent of the cut-irrelevant part of $\mathscr{H}_{\hat{\sigma}}$ and D_i is the d-hyperclause corresponding to C_i. We instantiate $R_i(\hat{\sigma})$ using θ and finally apply $(distr)$, as indicated in Section 6, to obtain a cut-free proof $\hat{\sigma}_i^{\theta}$ of $\theta(\mathscr{G}_i) \odot \theta(C_i)$.

To get a proof $\gamma[\hat{\sigma}]$ of a linked Skolem instance of the original hypersequent $\mathscr{H}$ (cf. Section 4) we replace each leaf node $\theta(C_i)$ of γ with the proof $\hat{\sigma}_i^{\theta}$ of $\theta(\mathscr{G}_i) \odot \theta(C_i)$, described above, and transfer the instances $\theta(\mathscr{G}_i)$ of cut-irrelevant formulas in $\mathscr{H}$ also to the inner nodes of γ in the obvious way, i.e., to regain correct applications of atomic cuts. As mentioned in Section 3, the remaining atomic cuts can easily be removed from $\gamma[\hat{\sigma}]$. The resulting proof is subjected to de-Skolemization as described in Theorem 3. This final step 7 of hyperCERES yields the desired cut-free proof of $\mathscr{H}$.

Example 2. We continue Example 1, where we have obtained the characteristic set of pairs $\{\langle \rho_1, \overset{\{1\}}{\Rightarrow} Q\rangle,\ \langle \rho_2, P(x) \overset{\{4\}}{\Rightarrow}\rangle,\ \langle \rho_3, Q \overset{\{2\}}{\Rightarrow} \mid \overset{\{3,5\}}{\Rightarrow}\rangle\}$ for the proof σ of the (trivially) Skolemized prenex hypersequent $Q \overset{\{1,2,4\}}{\Rightarrow} \exists y P(y) \mid P(c) \vee Q \overset{\{3,5\}}{\Rightarrow} Q$.

The obtained d-hyperclauses are in fact already hyperclauses. Moreover, one can immediately see that the hyperclauses $\overset{\{1\}}{\Rightarrow} Q$ and $Q \overset{\{2\}}{\Rightarrow} \mid \overset{\{3,5\}}{\Rightarrow}$ can be refuted by a one-step resolution derivation γ:

$$\dfrac{\overset{\{1\}}{\Rightarrow} Q \qquad Q \overset{\{2\}}{\Rightarrow} \mid \overset{\{3,5\}}{\Rightarrow}}{\overset{\{1,2\}}{\Rightarrow} \mid \overset{\{1,3,5\}}{\Rightarrow}}\,(res)$$

Note that $P(x) \overset{\{4\}}{\Rightarrow}$ and the corresponding reduced proof ρ_2 are redundant. In our case, γ is already ground. Therefore no substitution has to be applied to the reduced proofs ρ_1 and ρ_3. By replacing the two upper (d-)hyperclauses in γ with ρ_1 and ρ_3, respectively we obtain the desired proof $\gamma[\sigma]$ that only contains an atomic cut:

$$\dfrac{\dfrac{\dfrac{\dfrac{Q, P(c) \overset{\{2\}}{\Rightarrow} P(c) \mid \overset{\{3,5\}}{\Rightarrow}}{Q, P(c) \overset{\{2\}}{\Rightarrow} \exists y P(y) \mid \overset{\{3,5\}}{\Rightarrow}}\,(\exists\text{-}r) \qquad \dfrac{Q \overset{\{2\}}{\Rightarrow} \mid \overset{\{3,5\}}{\Rightarrow} Q}{Q \overset{\{2\}}{\Rightarrow} \mid P(c) \overset{\{3,5\}}{\Rightarrow} Q}\,(iw)\text{-}l}{Q \overset{\{2\}}{\Rightarrow} \exists y P(y) \mid P(c) \overset{\{3,5\}}{\Rightarrow} Q}\,(com) \qquad Q \overset{\{5\}}{\Rightarrow} Q}{Q \overset{\{2,4\}}{\Rightarrow} \exists y P(y) \mid P(c) \vee Q \overset{\{3,5\}}{\Rightarrow} Q}\,(\vee\text{-}l) \qquad Q \overset{\{1\}}{\Rightarrow} Q}{Q \overset{\{1,2,4\}}{\Rightarrow} \exists y P(y) \mid P(c) \vee Q \overset{\{3,5\}}{\Rightarrow} Q}\,(cut)$$

8 Final Remarks

The results of this paper are easily extendable to larger fragments **G**: (de-)Skolemization is sound already for intuitionistic logic **I** without positive occurrences of universal quantifiers, if an additional existence predicate is added [6]. Therefore hyperCERES applies after incooperation of the mentioned existence predicate. Other classes where Skolemization is sound for **I** are described by Mints [16].

The most interesting question however is whether hyperCERES can be extended to intuitionistic logic itself. Note that we obtain a calculus for **I** by dropping the communication rule from **HG**. It turns out that hyperCERES is applicable to the class of

(intuitionistic) hypersequents not containing negative occurrences of $\vee$ or positive occurrences of $\forall$, as the distribution rule $(distr)$ is still sound for this fragment of **I**. This fragment actually is an extension of the Harrop class [14] with weak quantifiers.

The extendability of hyperCERES to full intuitionistic logic depends on the development of an adequate (de-)Skolemization technique, together with a concept of parallelized resolution refutations, that takes into account the disjunctions of atoms at the right hand side of clauses without using $(distr)$.

From a more methodological viewpoint, it should be mentioned that hyperCERES uses the fact that 'negative information' can be treated classically in intermediate logics like **G**, and that cuts amount to entirely negative information in our approach. In this sense, global cut elimination, as presented in this paper, is more adequate for intermediate logics than stepwise reductions, which treat cuts as positive information.

References

1. Avron, A.: Hypersequents, Logical Consequence and Intermediate Logics for Concurrency. Annals of Mathematics and Artificial Intelligence 4, 225–248 (1991)
2. Avron, A.: The Method of Hypersequents in Proof Theory of Propositional Non-Classical Logics. In: Logic: From Foundations to Applications, pp. 1–32. Clarendon Press (1996)
3. Baaz, M., Ciabattoni, A.: A Schütte-Tait style cut-elimination proof for first-order Gödel logic. In: Egly, U., Fermüller, C. (eds.) TABLEAUX 2002. LNCS, vol. 2381, pp. 24–38. Springer, Heidelberg (2002)
4. Baaz, M., Ciabattoni, A., Fermüller, C.G.: Herbrand's Theorem for Prenex Gödel Logic and its Consequences for Theorem Proving. In: Nieuwenhuis, R., Voronkov, A. (eds.) LPAR 2001. LNCS, vol. 2250, pp. 201–216. Springer, Heidelberg (2001)
5. Baaz, M., Hetzl, S., Leitsch, A., Richter, C., Spohr, H.: CERES: An Analysis of Fürstenberg's Proof of the Infinity of Primes. Theoretical Computer Science (to appear)
6. Baaz, M., Iemhoff, R.: The Skolemization of existential quantifiers in intuitionistic logic. Ann. of Pure and Applied Logics 142, 269–295 (2006)
7. Baaz, M., Leitsch, A.: Cut-elimination and Redundancy-elimination by Resolution. J. Symb. Comput. 29(2), 149–177 (2000)
8. Baaz, M., Leitsch, A.: CERES in Many-Valued Logics. In: Baader, F., Voronkov, A. (eds.) LPAR 2004. LNCS, vol. 3452, pp. 1–20. Springer, Heidelberg (2005)
9. Baaz, M., Leitsch, A.: Towards a clausal analysis of cut-elimination. J. Symb. Comput. 41(3-4), 381–410 (2006)
10. Baaz, M., Leitsch, A., Zach, R.: Incompleteness of an infinite-valued first-order Gödel Logic and of some temporal logic of programs. In: Kleine Büning, H. (ed.) CSL 1995. LNCS, vol. 1092, pp. 1–15. Springer, Heidelberg (1996)
11. Baaz, M., Zach, R.: Hypersequents and the Proof Theory of Intuitionistic Fuzzy Logic. In: Clote, P.G., Schwichtenberg, H. (eds.) CSL 2000. LNCS, vol. 1862, pp. 187–201. Springer, Heidelberg (2000)
12. Chagrov, A., Zakharyaschev, M.: Modal Logic. Oxford University Press, Oxford (1997)
13. Hájek, P.: Metamathematics of Fuzzy Logic. Kluwer, Dordrecht (1998)
14. Harrop, R.: Concerning formulas of the types $A \supset B \vee C, A \supset (\exists x)B(x)$ in intuitionistic formal systems. J. Symbolic Logic 25, 27–32 (1960)
15. Leitsch, A.: The Resolution Calculus. Springer, Heidelberg (1997)
16. Mints, G.: The Skolem method in intuitionistic calculi. Proc. Inst. Steklov. 121, 73–109 (1974)

17. Orevkov, V.P.: Lower Bounds for Increasing Complexity of Derivations after Cut Elimination. J. Soviet Mathematics, 2337–2350 (1982)
18. Schütte, K.: Beweistheorie. Springer, Heidelberg (1960)
19. Statman, R.: Lower bounds on Herbrand's theorem. Proc. of the Amer. Math. Soc. 75, 104–107 (1979)
20. Tait, W.W.: Normal derivability in classical logic. The Syntax and Semantics of infinitary Languages LNM 72, 204–236 (1968)
21. Troelstra, A.S., Schwichtenberg, H.: Basic Proof Theory, 2nd edn., Cambridge (2000)
22. Takeuti, G., Titani, T.: Intuitionistic fuzzy logic and intuitionistic fuzzy set theory. J. Symbolic Logic 49, 851–866 (1984)

Focusing Strategies in the Sequent Calculus of Synthetic Connectives

Kaustuv Chaudhuri

INRIA Saclay - Île-de-France
Kaustuv.Chaudhuri@inria.fr

Abstract. It is well-known that focusing striates a sequent derivation into phases of like polarity where each phase can be seen as inferring a synthetic connective. We present a sequent calculus of synthetic connectives based on neutral proof patterns, which are a syntactic normal form for such connectives. Different focusing strategies arise from different polarisations and arrangements of synthetic inference rules, which are shown to be complete by synthetic rule permutations. A simple generic cut-elimination procedure for synthetic connectives respects both the ordinary focusing and the maximally multi-focusing strategies, answering the open question of cut-admissibility for maximally multi-focused proofs.

1 Introduction

The story of focusing has been told several times since Andreoli [3] with essentially the same construction. The inference rules of the sequent calculus divide into two groups: invertible and non-invertible, and, during proof search, the invertible rules can be eagerly applied until only non-invertible rules remain. Then, one connective is selected for *focus* and a maximal sequence of non-invertible rules are applied until invertible rules become available again. Focused proof search alternates between these two phases, invertible and non-invertible, or *negative* and *positive*, until no unproven goals remain. Furthermore, the principal formulas of the positive and negative rules, themselves called positive and negative, are perfectly dual, evoking a number of dualities that have recently been explained via focusing; a short list includes: call-by-value (positive) dual to call-by-name (negative) [16]; the Q-protocol (positive) dual to the T-protocol (negative) [14]; the proponent (positive) dual to the opponent (negative) [2]; and forward-chaining (positive) dual to backward-chaining (negative) [7] (see [18] for a survey).

Proof theoretically, the innovation of focusing is not its operational interpretation, however, but the derived notion of a *synthetic connective*. If the operand of a positive (resp. negative) connective is itself positive (resp. negative), then the two connectives fuse into a larger positive (resp. negative) connective; this fusion eventually synthesises connectives whose operands have the opposite polarity and whose internal structure does not matter; thus, $- \otimes (- \otimes -)$ and $(- \otimes -) \otimes -$ are essentially the same ternary positive synthetic connective when applied to three negative operands. A focused derivation amounts to a derivation using synthetic inference rules for such synthetic connectives.

But are synthetic connectives true connectives, and can one construct a traditional sequent calculus based on synthetic inferences? A first approximation to an answer is

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 467–481, 2008.

to note that the focused sequent calculus admits identity and cut-elimination [7,14], so *provability*-wise there is no problem with synthetic inferences. However, if one looks at the *proofs* themselves, the question gets more clouded. Ordinary unfocused rules of like polarity freely permute with each other (as long as not prohibited by the subformula relation), but synthetic positive rules never do. Indeed, the question of positive-positive permutation is meaningless in the standard presentation of focusing because the neighbours of a positive phase are negative. Yet, a positive-positive permutation is not a dubious concept: consider the sequent $\vdash a^\perp \otimes \perp, b^\perp \otimes \perp, a, b, \mathbf{1}$, for instance, where indeed the two $\otimes$ rules are non-interfering and permute. In game-theoretic terms, the equivalent $\otimes$ moves are *truly concurrent*; however, the focused sequent calculus can only represent a serialisation, which has been a long standing criticism of focusing *qua* syntax [1]. This limitation was partially removed in [5] by the use of *multi-focusing* to represent the truly concurrent foci, but in that work the question of synthetic permutation was answered by discarding one half of the synthetic connectives (the negatives) in their entirety– very unsatisfactory!

The second break in the proof theory of synthetic derivations comes from the explicit polarity switches or *delays*. These switching connectives are commonly used to define a so-called *polarised linear logic* [10,12]; in fact, careful use of switches allows one to mimic "strongly focusing" calculi like LJQ and LJT in a general focusing framework [14]. This connective, however, has an incarnation only in the polarised world; the unfocused calculus cannot detect it (*e.g.*, by cut-reduction). Why should one countenance the invention of new connectives from one's choice of proofs?

In this paper we propose an answer to both questions by paring focusing down into three orthogonal concepts, each involving choices specific forms, that are usually confused together: *neutral expressions*, *proof patterns*, and *focusing strategies*. To be concrete, we limit ourselves to propositional MALL, although the construction itself is as general as focusing. The central component, proof patterns, is a technical device used to construct static synthetic inference rules by explicitly representing a normal form of the branching *search tree* for a synthetic connective. Synthetic identity, synthetic cut-elimination, and synthetic permutations can be defined by simple analysis of these static proof patterns. The other two concepts are dynamic. Neutral expressions represent the polarisation of a connective by recording the phase changes with the switch; however, the switches themselves are not connectives but part of the dynamics of the proof: prolonging phases as long as possible, as in ordinary focusing, is a *maximal* polarisation, while switching phases always, as in unfocused calculi, is a *minimal polarisation*. The third concept, focusing strategies, defines recipes for applying the synthetic inference rules; ordinary focusing is a negative-eager strategy, and maximal multi-focusing [5] is a positive-eager strategy. The generic synthetic cut elimination preserves both these strategies using a priority assignment to cut permutations.

Our main departure from ordinary focusing is an abolishment of the regimented phase alternation. Relaxed phase alternation is a recent innovation in focusing, but it has appeared at least three times before: first in [15] where the *focalisation graph* exposed several roots that could be simultaneously or sequentially focused on, second in [9] where a neutral game is forced to pick multiple positive foci to maintain parity with the multiple negative "foci", and third in [5] which recovers a limited form of

permutative canonicity by requiring the foci to be as large as possible. Our presentation is reminiscent of several related exegeses of focusing: the generic cut-elimination is present in ludics [10], even though it is phase alternating (and monistic, which our presentation decidedly isn't); proof patterns are a generalisation of a similar construct in the CU calculus [18]; neutral expressions are present in [9], although their use there was to define a neutral game that grows a dual pair of mutually normalising derivations.

This paper is organised as follows. In sec. 2, neutral expressions and proof patterns are formally introduced and the sequent calculus of synthetic inferences is defined. The key identity (theorem. 11) and cut (theorem. 12) theorems are proved. In sec. 3, permutation is defined for synthetic connectives. Sec. 4 introduces focusing strategies and sketches the two main variants: ordinary and maximally multi-focused.

2 A Sequent Calculus of Synthetic Connectives

This section will reconstruct a cut-free sequent calculus, called MALL-S, of synthetic connectives and inference rules for propositional multiplicative-additive linear logic. MALL is selected for simplicity and clarity; it contains the important features of focusing without the distraction of polarising the exponentials. We shall adopt a polarised syntax similar to [13], but polarised propositions will be seen as dual interpretations of neutral expressions.

Definition 1. Neutral expressions*, written E, F, etc., have the following syntax.*

$$E, F, \ldots ::= a \mid E \times F \mid 1 \mid E + F \mid 0 \mid \updownarrow E$$

Here, a represents an atomic proposition with unassigned polarity. The $\updownarrow$ *operator represents an explicit switch of polarities.*

Definition 2 (polarisation). *A polarised proposition is defined as either a positive or a negative polarisation of a neutral expression, given respectively by the polarisation functions* $\langle - \rangle^+$ *and* $\langle - \rangle^-$*:*

$$(a)^+ = a^+ \quad (a)^- = a^-$$

$$\langle E \times F \rangle^+ = \langle E \rangle^+ \otimes \langle F \rangle^+ \quad \langle 1 \rangle^+ = \mathbf{1} \quad \langle E \times F \rangle^- = \langle E \rangle^- \parr \langle F \rangle^- \quad \langle 1 \rangle^- = \bot$$

$$\langle E + F \rangle^+ = \langle E \rangle^+ \oplus \langle F \rangle^+ \quad \langle 0 \rangle^+ = \mathbf{0} \quad \langle E + F \rangle^- = \langle E \rangle^- \,\&\, \langle F \rangle^- \quad \langle 0 \rangle^- = \top$$

$$\langle \updownarrow E \rangle^+ = \langle E \rangle^- \quad \langle \updownarrow E \rangle^- = \langle E \rangle^+$$

Here, a^+ *(resp.* a^-*) refers to a positively (resp. negatively) polarised atom. We write* $\langle E \rangle^\pm$ *to refer to either* $\langle E \rangle^+$ *or* $\langle E \rangle^-$*, and* $\langle E \rangle^\mp$ *to refer to its dual polarisation.*

Note that polarisation is an injection: infinitely many expressions can polarise to the same proposition by means of repeated "administrative" $\updownarrow$s. Nevertheless, all MALL propositions are either positive or negative polarisations of some expression. This restriction will need to be relaxed when moving to non-linear logics such as classical logic where the propositional connectives have ambiguous polarities. Note also that the representation of & as a polarisation of a sum (+) differs starkly from the popular view of & as a "conjunction", possibly because the two polarised interpretations of the classically ambiguous $\wedge$ are $\otimes$ and & [14]. Our view of & as a "sum" (and $\parr$ as a "product") is supported by distributivity: $A \parr (B \,\&\, C) \equiv (A \parr B) \,\&\, (A \parr C)$.

Indeed, the rules of the calculus will be given not for polarised neutral expressions but for an associated unique *proof pattern* that reorganises the expression into a disjunctive normal form up to the atoms or the polarity switches, using distributivity to move the +s to the surface through the ×s. This reorganisation will generally repeat a sub-expression—for example, $a \times (b + c)$ is reorganised to $(a \times b) + (a \times c)$, repeating a—but *will not* duplicate any sub-derivations because of the $\updownarrow$ guards. For instance, the duplication of the $\vdash B$ in the following derivation is syntactically prohibited.

$$\dfrac{\dfrac{\dfrac{\vdash A, C \qquad \vdash A, D}{\vdash A, C \,\&\, D}\,\&}{\vdash A \otimes B, C \,\&\, D}\,\otimes}{\vdash (A \otimes B) \parr (C \,\&\, D)}\,\parr \quad \equiv \quad \dfrac{\dfrac{\dfrac{\vdash A, C \qquad \vdash B}{\vdash A \otimes B, C}\,\otimes \qquad \dfrac{\vdash A, D \qquad \vdash B}{\vdash A \otimes B, D}\,\otimes}{\vdash A \otimes B, C \,\&\, D}\,\&}{\vdash (A \otimes B) \parr (C \,\&\, D)}\,\parr$$

The propositions $A \otimes B$ and $C \,\&\, D$ are of opposite polarities, so we only observe the equivalence $\updownarrow(E \times F) \times (G + H) \equiv (\updownarrow(E \times F) \times G) + (\updownarrow(E \times F) \times H)$ (for suitable E, F, G and H), both of which have identical sub-derivations after the outer negative phase.

Definition 3 (proof patterns). Product patterns *(π) and* sum patterns *(σ) are generated by the following grammars:*

$$\pi ::= \hat{E} \mid 1 \mid \pi_1 \cdot \pi_2 \qquad\qquad \sigma ::= \pi \mid 0 \mid \sigma_1 + \sigma_2$$

where $\hat{E}$ is either an atom or of the form $\updownarrow E$. The structures $\langle \Pi, \cdot, 1\rangle$ and $\langle \Xi, +, 0\rangle$ are commutative monoids, where Π is the set of product patterns and Ξ the set of sum patterns. A product pattern will always be depicted in its normal form $\prod_{i\in I} \hat{E}_i$, and a sum pattern similarly as $\sum_{i\in I} \pi_i$, where I is a finite index set. The unqualified term "proof pattern" will refer to sum patterns.

Notation 4. *We write $\Downarrow\left\langle \hat{E} \right\rangle^{\pm}$ for $(a)^{\pm}$ if $\hat{E} = a$ and for $\langle F \rangle^{\mp}$ if $\hat{E} = \updownarrow F$.*

Abstractly, a proof pattern $\sum_{i\in I} \pi_i$ represents the proof search tree for a synthetic connective. In the positive interpretation of the connective, the outer sum represents the disjunctive ($\oplus$) choices, while each π_i represents the multiplicative ($\otimes$) structure. In the negative interpretation, the outer sum represents the alternatives (&), while the inner product represents the sequent structure ($\parr$).

Definition 5. *Given two patterns $\sigma = \sum_{i\in I} \pi_i$ and $\sigma' = \sum_{j\in J} \pi'_j$, their* product*, written $\sigma \times \sigma'$ is the pattern $\sum_{i\in I} \sum_{j\in J} \pi_i \cdot \pi'_j$.*

Fact 6. *$\langle \Xi, +, 0, \times, 1\rangle$ is a commutative semiring.* □

Every neutral expression has a corresponding pattern derived simply by treating the expression constructors as these semiring operators. Precisely, we can define a judgement $\sigma \Vdash E$ with the following rules:

$$\frac{}{a \Vdash a} \qquad \frac{}{\updownarrow E \Vdash \updownarrow E} \qquad \frac{\sigma \Vdash E \qquad \sigma' \Vdash F}{\sigma \times \sigma' \Vdash E \times F} \qquad \frac{}{1 \Vdash 1} \qquad \frac{\sigma \Vdash E \qquad \sigma' \Vdash F}{\sigma + \sigma' \Vdash E + F} \qquad \frac{}{0 \Vdash 0}$$

For example, $(a \cdot c) + (a \cdot \updownarrow E) + c + \updownarrow E \Vdash (a + 1) \times (c + \updownarrow E)$.

Before proceeding further, we note that a similar notion of "pattern" has been developed in the realm of focused lambda calculi in the system CU [18,17]. The differences are as follows: we define patterns for both positive and negative propositions and

furthermore represent both the product and the sum structure; patterns in CU, on the other hand, are defined only for the positive propositions and keep just the products, forgetting one half of each sum. CU patterns are therefore a slice of the disjunctive normal form—indeed, the disjunctive normal forms cannot be computed at all unless the sums are represented—which necessitates quantification over (their equivalent of) the $\Vdash$ judgement to recover the full sum. This quantification makes the synthetic rules in CU higher-order. (This is most likely by design, as CU is intended as a logical explanation for higher-order abstract syntax.)

Definition 7 (contexts and sequents)

- *A* context *is a finite multiset of expressions annotated with polarities,* $(E_1)^{\pm}, \ldots, (E_n)^{\pm}$, *written* Δ. *Two contexts that differ only on* $(E)^{\pm}$ *and* $(F)^{\pm}$ *with* $E \neq F$ *are considered different even if* $(E)^{\pm} = (F)^{\pm}$ *(defn. 2).*
- *A* sequent $\vdash \Delta$, *where* Δ *is a context, is a judgement that the context* Δ *is linearly contradictory. The form* $\vdash_{\xi} \Delta$ *is used to indicate that* ξ *is a derivation of* $\vdash \Delta$.
- *A* focused sequent *is a structure of the form* $\vdash \Delta \,;\, (\pi)^{\pm}$ *where* Δ *is a context and* $(\pi)^{\pm}$ *is a product pattern* π *annotated with a polarisation.*

Definition 8. *Let* $\mathbf{D} = \langle \Delta_i \rangle_{i \in 1..n}$ *be a list of contexts. Then,*

1. $\bigotimes \mathbf{D}$ *stands for the list of sequents* $\vdash \Delta_1, \ldots, \vdash \Delta_n$.
2. $\parr \mathbf{D}$ *stands for the sequent* $\vdash \Delta_1, \ldots, \Delta_n$.

For non-atomic principal formulas, we define the synthetic rules on the proof pattern of the underlying neutral expression. The outer sum in the pattern represents an enumeration of choices. If the corresponding proposition were positively polarised, then it represents a disjunction of choices of which only one needs to be taken. For a negatively polarised proposition it represents an alternation of choices, *all* of which must be taken. The *bottom half* of the two synthetic rules thus looks as follows:

$$\frac{\sum_{i \in I} \pi_i \Vdash E \qquad \exists u \in I.\ \vdash \Delta \,;\, (\pi_u)^{+}}{\vdash \Delta, (E)^{+}}\ \mathrm{P}\downarrow \qquad \frac{\sum_{i \in I} \pi_i \Vdash E \qquad \forall u \in I.\ \vdash \Delta \,;\, (\pi_u)^{-}}{\vdash \Delta, (E)^{-}}\ \mathrm{N}\downarrow$$

In each case, we obtain a number of focused sequents of the form $\vdash \Delta \,;\, (\pi)^{\pm}$. If the polarisation is positive, i.e., the product pattern represents a $\otimes$, then the context Δ must be distributed into the components of the product. This *demultiplexion* operation is defined by a ternary relation.

Definition 9 (demultiplexion). *Given a context* Δ *and a product pattern* $\pi = \prod_{i \in 1..k} \hat{E}_i$, *a* demultiplexion *of* Δ along $(\pi)^{\pm}$ *produces a list of contexts* $\mathbf{D} = \Delta_1 \,;\, \cdots \,;\, \Delta_k$, *written* $\Delta \,;\, (\pi)^{+} \gg \mathbf{D}$, *generated by the following rules:*

$$\frac{\Delta \,;\, (\pi)^{+} \gg \mathbf{D}}{\Delta, (a)^{-} \,;\, (\pi \cdot a)^{+} \gg \mathbf{D} \,;\, (a)^{-}, (a)^{+}}$$

$$\frac{\Delta \,;\, (\pi)^{-} \gg \mathbf{D}}{\Delta, \Delta' \,;\, (\pi \cdot a)^{-} \gg \mathbf{D} \,;\, \Delta', (a)^{-}} \qquad \frac{\Delta \,;\, (\pi)^{\pm} \gg \mathbf{D}}{\Delta, \Delta' \,;\, (\pi \cdot \updownarrow E)^{\pm} \gg \mathbf{D} \,;\, \Delta', (E)^{\mp}}$$

The upper half of the positive rule, P↑, is then obvious: we select a demultiplexion of the context along the positive pattern and interpret every context in it as a sequent.

$$\frac{\exists \mathbf{D} : (\Delta\,;(\pi)^+ \gg \mathbf{D}).\ \bigotimes \mathbf{D}}{\vdash \Delta\,;(\pi)^+}\ \mathrm{P}\uparrow$$

Somewhat surprisingly, the upper half of the negative rule, N↑, can be written analogously:

$$\frac{\forall \mathbf{D} : (\Delta\,;(\pi)^- \gg \mathbf{D}).\ \parr \mathbf{D}}{\vdash \Delta\,;(\pi)^-}\ \mathrm{N}\uparrow$$

The demultiplexion used to construct $\mathbf{D}$ is simply undone by the $\parr$ operator. It therefore does not matter how the demultiplexion is done, and there is always a way to demultiplex along a negative pattern (for example, all of the context can be "sent" to the first element of the product pattern, if one exists). The premise of the N↑ rule is therefore uniquely determined.

We shall henceforth ignore the halves of the rules and just consider the combined rules P and N, each of which is a polarisation of the following neutral rule (where E is non-atomic and $\sum_{i\in I} \pi_i \Vdash E$):

$$\frac{Qu \in I.\ Q\mathbf{D} : (\Delta\,;(\pi_u)^p \gg \mathbf{D}).\ \bigcirc \mathbf{D}}{\vdash \Delta, (E)^p}\ \mathrm{R}(p, Q, \bigcirc)$$

In the positive interpretation $\mathrm{P} = \mathrm{R}(+, \exists, \bigotimes)$, there is one premise corresponding to each element of a demultiplexion of the context along a product pattern; in the negative interpretation $\mathrm{N} = \mathrm{R}(-, \forall, \parr)$, there is one premise for each element of the outer sum pattern. The rules have been written in this way to highlight the precise duality of their premises.

It is important to note that the use of the $\forall$ and $\exists$ in the rules is merely a notational device. As the pattern that corresponds to an expression is statically known, we actually have instances of the P and N rules specialised to the index sets of these statically known patterns. Thus, the P and N rules are "first-order": they do not depend on reasoning in the meta-language.

There is one additional synthetic rule for atomic propositions:

$$\frac{}{\vdash (a)^+, (a)^-}\ \mathrm{I}$$

This rule is neither positive, nor negative, and can only be applied at the leaves of a derivation. As usual, it is only defined for atoms, but it can be proved for arbitrary expressions. This is a syntactic completeness theorem that is usually called the *identity principle*. Its proof is almost immediate in MALL-S.

Notation 10. *We shall adopt the following notational shorthands: $\{\Delta_i\}_{i\in I}$ for the multiset union of the Δ_i, $\langle \Gamma_i \rangle_{i\in I}$ for the list of contexts Γ_i, and $\langle \vdash \Gamma_i \rangle_{i\in I}$ for the list of premises $\vdash \Gamma_i$.*

Theorem 11 (identity principle). *The sequent $\vdash (E)^+, (E)^-$ is derivable for any E.*

Proof. We reason by induction on the structure of the proof pattern for E. The atomic case follows simply by I. For the non-atomic cases, suppose $\sum_{i\in I}\prod_{j\in J_i}\hat{E}_{ij} \Vdash E$. We have:

$$\dfrac{\forall u \in I. \quad \dfrac{\left\langle \vdash \Downarrow\left(\hat{E}_{uj}\right)^+, \Downarrow\left(\hat{E}_{uj}\right)^- \right\rangle_{j\in J_u}}{\vdash (E)^+, \left\{\Downarrow\left(\hat{E}_{uj}\right)^-\right\}_{j\in J_u}}\ \mathrm{P}}{\vdash (E)^+, (E)^-}\ \mathrm{N}$$

because

$$\left\{\Downarrow\left(\hat{E}_{uj}\right)^-\right\}_{j\in J_u}; \left(\prod_{j\in J_u}\hat{E}_{uj}\right)^+ \Vdash \left\langle \Downarrow\left(\hat{E}_{uj}\right)^+, \Downarrow\left(\hat{E}_{uj}\right)^- \right\rangle_{j\in J_u}.$$

We then use the induction hypothesis on the sequents of the form $\vdash \Downarrow\left(\hat{E}_{uj}\right)^+, \Downarrow\left(\hat{E}_{uj}\right)^-$, which contain strictly smaller expressions. □

For syntactic soundness, we turn to admissibility of the following *cut* rule:

$$\dfrac{\vdash \Delta, (E)^+ \qquad \vdash \Gamma, (E)^-}{\vdash \Delta, \Gamma}\ \mathrm{C}$$

Theorem 12 (cut elimination). *The C rule is admissible*

Proof. We shall prove this by first admitting C as an inference rule and then eliminating it by (non-deterministically) rewriting it out of a proof that uses it. This rewrite $\longrightarrow$ is generated from the following cases.

- Initial cuts, where one of the premises is derived from I. In these cases, the elimination is trivial because we can just drop the C and the initial premise.
- Principal cuts, where the cut-expression is principal in P and N in the two premises. Suppose $\sum_{i\in I}\pi_i \Vdash E$, $\pi_i = \prod_{j\in J_i}\hat{E}_{ij}$ and $u \in I$, such that:

$$\dfrac{\dfrac{\left\langle \vdash_{\xi(j)} \Gamma_j, \Downarrow\left(\hat{E}_{uj}\right)^+ \right\rangle_{j\in J_u}}{\vdash \{\Gamma_j\}_{j\in J_u}, (E)^+}\ \mathrm{P} \qquad \dfrac{\forall i \in I.\ \vdash_{\zeta(i)} \Delta, \left\{\Downarrow\left(\hat{E}_{ij}\right)^-\right\}_{j\in J_i}}{\vdash \Delta, (E)^-}\ \mathrm{N}}{\vdash \{\Gamma_j\}_{j\in J_u}, \Delta}\ \mathrm{C}$$

  Let $\phi : 1..n \to J_u$ be a bijection. We rewrite the above cut as follows:

$$\dfrac{\vdash_{\xi(\phi(1))} \Gamma_{\phi(1)}, \Downarrow\left(\hat{E}_{u\phi(1)}\right)^+ \qquad \dfrac{\vdash_{\xi(\phi(2))} \Gamma_{\phi(2)}, \Downarrow\left(\hat{E}_{u\phi(2)}\right)^+ \qquad \dfrac{\vdots \qquad \dfrac{\vdots}{\vdash_{\xi(\phi(n))} \Gamma_{\phi(n)}, \Downarrow\left(\hat{E}_{u\phi(n)}\right)^+ \qquad \vdash_{\zeta(u)} \Delta, \left\{\Downarrow\left(\hat{E}_{u\phi(k)}\right)^-\right\}_{k\in 1..n}}\ \mathrm{C}}{\vdash \Gamma_{\phi(n)}, \Delta, \left\{\Downarrow\left(\hat{E}_{u\phi(k)}\right)^-\right\}_{k\in 1..n-1}}\ \mathrm{C}}{\vdash \{\Gamma_{\phi(k)}\}_{k\in 2..n}, \Delta, \Downarrow\left(\hat{E}_{i\phi(1)}\right)^-}\ \mathrm{C}}{\vdash \{\Gamma_{\phi(k)}\}_{k\in 2..n}, \Delta}\ \mathrm{C}$$

  Each instance of C is now on a strictly smaller cut expression.
- Commutative cuts, where the cut-expression is not principal in one derivation. In each of the following two cases, we suppose that $\sum_{i\in I}\prod_{j\in J_i}\hat{F}_{ij} \Vdash F$, $u \in I$ and

$v \in J_u$. The two cases of the rewrite are named [PCC] and [NCC] for positive and negative commutative cuts respectively.

$$\dfrac{\dfrac{\bigotimes \mathbf{D} \qquad \vdash_\xi \Delta_v, (E)^\pm, \Downarrow\left(\hat{F}_{uv}\right)^+ \qquad \bigotimes \mathbf{D}'}{\vdash \{\Delta_j\}_{j\in J_u}, (F)^+, (E)^\pm}\,\mathrm{P} \qquad \vdash_\zeta \Gamma, (E)^\mp}{\vdash \{\Delta_j\}_{j\in J_u}, (F)^+, \Gamma}\,\mathrm{C} \quad \longrightarrow_{[\mathrm{PCC}]}$$

$$\dfrac{\bigotimes \mathbf{D} \qquad \dfrac{\vdash_\xi \Delta_v, (E)^\pm, \Downarrow\left(\hat{F}_{uv}\right)^+ \qquad \vdash_\zeta \Gamma, (E)^\mp}{\vdash \Delta_v, (E)^\pm, \Downarrow\left(\hat{F}_{uv}\right)^+, \Gamma}\,\mathrm{C} \qquad \bigotimes \mathbf{D}'}{\vdash \{\Delta_j\}_{j\in J_u}, (F)^+, \Gamma}\,\mathrm{P}$$

$$\dfrac{\dfrac{\forall i \in I.\ \vdash_\xi \Delta, (E)^\pm, \left\{\Downarrow\left(\hat{F}_{ij}\right)^-\right\}_{j\in J_i}}{\vdash \Delta, (F)^-, (E)^\pm}\,\mathrm{P} \qquad \vdash_\zeta \Gamma, (E)^\mp}{\vdash \Delta, (F)^-, \Gamma}\,\mathrm{C} \quad \longrightarrow_{[\mathrm{NCC}]}$$

$$\dfrac{\forall i \in I. \qquad \dfrac{\vdash_\xi \Delta, (E)^\pm, \left\{\Downarrow\left(\hat{F}_{ij}\right)^-\right\}_{j\in J_i} \qquad \vdash_\zeta \Gamma, (E)^\mp}{\vdash \Delta, \Gamma, \left\{\Downarrow\left(\hat{F}_{ij}\right)^-\right\}_{j\in J_i}}\,\mathrm{C}}{\vdash \Delta, (F)^-, (E)^\pm}\,\mathrm{P}$$

In these cases the instance of C on the right is on a smaller derivation. □

The cut-elimination proof above is remarkable for several reasons. First, it is a generic argument that is independent of any logical connective. Second, it is obviously correct for each cut can be seen to be smaller by inspection.[1] Lastly, it is compact: there is no important detail missing in the proof. To be sure, these remarkable properties are also observed in CU [18,17], but in MALL-S, because the rules are first-order, we do not need to depend on the meta language for the proof of coverage.

As already mentioned, a key distinguishing feature of MALL-S from other focusing systems such as LLF [3] or CU is that the positive and negative rules do not alternate. In this sense, it would be a mistake to call MALL-S a "focusing" system, so we cannot prove MALL-S complete with respect to the unfocused MALL (rules in fig. 1) by citation. Fortunately, we can easily recover the unfocused rules by selecting a suitably minimal polarisation.

Definition 13. *The* minimal polarisation $\lfloor - \rfloor$ *is given inductively as follows:*

$$\lfloor a \rfloor = a \qquad \lfloor a^\perp \rfloor = a$$

$$\lfloor A \otimes B \rfloor = \lfloor A \rfloor^- \times \lfloor B \rfloor^- \qquad \lfloor \mathbf{1} \rfloor = 1 \qquad \lfloor A \parr B \rfloor = \lfloor A \rfloor^+ \times \lfloor B \rfloor^+ \qquad \lfloor \bot \rfloor = 1$$

$$\lfloor A \oplus B \rfloor = \lfloor A \rfloor^- + \lfloor B \rfloor^- \qquad \lfloor \mathbf{0} \rfloor = 0 \qquad \lfloor A \,\&\, B \rfloor = \lfloor A \rfloor^+ + \lfloor B \rfloor^+ \qquad \lfloor \top \rfloor = 0$$

$$\lfloor P \rfloor^- = \updownarrow\updownarrow \lfloor P \rfloor \qquad \lfloor N \rfloor^- = \updownarrow \lfloor N \rfloor \qquad \lfloor P \rfloor^+ = \updownarrow \lfloor P \rfloor \qquad \lfloor N \rfloor^+ = \updownarrow\updownarrow \lfloor N \rfloor$$

Here P (resp. N) refers to any positive (resp. negative) proposition.

[1] This proof does get more complex if, in some extension of MALL-S, the index sets are infinite, because in that case the result of eliminating a principal cut would be a derivation of infinite depth.

$$\frac{}{\vdash a, a^{\perp}}\ \mathrm{I} \qquad \frac{\vdash \Gamma, A \qquad \vdash \Delta, B}{\vdash \Gamma, \Delta, A \otimes B}\ \otimes \qquad \frac{}{\vdash \mathbf{1}}\ \mathbf{1} \qquad \frac{\vdash \Delta, A, B}{\vdash \Delta, A \parr B}\ \parr \qquad \frac{\vdash \Delta}{\vdash \Delta, \perp}\ \perp$$

$$\frac{\vdash \Delta, A_i}{\vdash \Delta, A_1 \oplus A_2}\ \oplus \qquad \text{no } \mathbf{0} \qquad \frac{\vdash \Delta, A \qquad \vdash \Delta, B}{\vdash \Delta, A \,\&\, B}\ \& \qquad \frac{}{\vdash \Delta, \top}\ \top$$

Fig. 1. MALL rules

Note that the polarisation of every strict subformula uses at least one administrative $\updownarrow$ switch, and that $(\lfloor P \rfloor)^+ = P$ and $(\lfloor N \rfloor)^- = N$. The rules of the ordinary MALL calculus then reappear as MALL-S rules for these minimal polarisations.

Theorem 14 (completeness of MALL-S). *If Δ is a context of unpolarised propositions, let $\lfloor \Delta \rfloor$ represent that context that replaces every positive $P \in \Delta$ with $(\lfloor P \rfloor)^+$ and every negative $N \in \Delta$ with $(\lfloor N \rfloor)^-$. If $\vdash \Delta$ in MALL, then $\vdash \lfloor \Delta \rfloor$ in MALL-S.*

Proof (sketch). By induction on the structure of the given MALL derivation.

Case of I: the I rules in MALL and MALL-S are identical.

Case of $\otimes$: Consider $\lfloor P \otimes N \rfloor = \lfloor P \rfloor^- \times \lfloor N \rfloor^- = \updownarrow\updownarrow \lfloor P \rfloor \times \updownarrow \lfloor N \rfloor$. Its proof pattern is just $\updownarrow\updownarrow \lfloor P \rfloor \cdot \updownarrow \lfloor N \rfloor$, so its derivation is:

$$\dfrac{\dfrac{\dfrac{}{\vdash \Delta_1, (\lfloor P \rfloor)^+}\ \text{i.h.}}{\vdash \Delta_1, (\updownarrow \lfloor P \rfloor)^-}\ \mathrm{N} \qquad \dfrac{}{\vdash \Delta_2, (\lfloor N \rfloor)^-}\ \text{i.h.}}{\vdash \Delta_1, \Delta_2, (\updownarrow\updownarrow \lfloor P \rfloor \times \updownarrow \lfloor N \rfloor)^+}\ \mathrm{P}$$
$$=$$
$$\vdash \Delta_1, \Delta_2, (\lfloor P \otimes N \rfloor)^+$$

because $\Delta_1, \Delta_2 \,;\, (\updownarrow\updownarrow \lfloor P \rfloor \times \updownarrow \lfloor N \rfloor)^+ \Vdash \Delta_1, (\updownarrow \lfloor P \rfloor)^- \,;\, \Delta_2, (\lfloor N \rfloor)^-$. We thus obtain the $\otimes$ rule for $P \otimes N$ in MALL. This characteristic case shows the way the induction works for subformulas of the same and opposite polarities, and the remaining cases are similar. □

Theorem 14 shows that the ordinary unfocused MALL is recoverable in MALL-S by picking specific polarisations. This is a strong indication that synthetic rules with delays are a more primitive notion than the usual binary connectives, an observation already made in the genesis of ludics [10], but not well appreciated outside a certain section of the proof theory community.

On the other hand, theorem. 14 does not show completeness for other polarisations. In ordinary (unpolarised) focusing [3] the polarity of the rules matches the natural polarity of the principal propositions. This suggests a maximal polarisation of the ordinary connectives that contains no administrative switches.

Definition 15. *E is a* maximal polarisation *for A if $A = (E)^{\pm}$ and E contains no subexpressions of the form $\updownarrow\updownarrow E'$. We write $\lceil A \rceil$ to refer to the unique maximal polarisation of A because of the following trivial fact.*

Fact 16. *If E and F are maximal polarisations for A, then $E = F$.* □

Lemma 17. $\vdash (\lfloor P \rfloor)^-, (\lceil P \rceil)^+$ *and* $\vdash (\lfloor N \rfloor)^+, (\lceil N \rceil)^-$ *are derivable in* MALL-S.

Proof (Sketch). Replay the proof of theorem. 11, but in one half of the reduction replay the maximally polarised synthetic rule with many minimally polarised synthetic rules. □

Minimal polarisations can thus be used to simulate maximal polarisations, directly giving the key focalisation result by an appeal to synthetic cuts.

Corollary 18 (Focalisation). *If Δ is a context of unpolarised propositions, let $\lceil \Delta \rceil$ represent that context that replaces every positive $P \in \Delta$ with $(\lceil P \rceil)^+$ and every negative $N \in \Delta$ with $(\lceil N \rceil)^-$. If $\vdash \Delta$ in* MALL*, then $\vdash \lceil \Delta \rceil$ in* MALL-S.

Proof. By theorem. 14, $\vdash \lfloor \Delta \rfloor$ is provable in MALL-S. Cut (theorem. 12) every $(\lfloor P \rfloor)^+$ and $(\lfloor N \rfloor)^-$ in $\vdash \lfloor \Delta \rfloor$ with a proof of $\vdash (\lfloor P \rfloor)^-, (\lceil P \rceil)^+$ or $\vdash (\lfloor N \rfloor)^+, (\lceil N \rceil)^-$ (lem. 17). □

Proofs of the focalisation result using cut-elimination have also been attempted in [7,14], but their proofs tend to be considerably more complex than the one presented here, partly because they are based on a more traditional formulation of focusing, but also because their proofs attempt to simulate the unfocused rules directly with cut. Our approach of simulating the unfocused rules with a different polarisation and then cutting them out *ex post facto* is a more perspicuous decomposition of the focalisation result.

3 Synthetic Permutations

In this section, we investigate the matter of permutations of the synthetic inference rules P and N. If the synthetic sequent calculus is to be seen as a generalisation of the ordinary calculus, then it is essential to define the corresponding generalisations of the binary permutations [11]. We write r_1/r_2 as a type of permutation where the rule(s) r_1 is (are) used immediately above r_2 and the result of the permutation moves (possibly with replication) r_2 above a single instance of r_1 without affecting the rest of the derivation. Such permutations are familiar from the ordinary (unfocused) logic; for instance the permutation $\otimes/\&$ in MALL is the following reordering:

$$\dfrac{\dfrac{\vdash_\xi \Gamma, A \qquad \vdash_\zeta \Delta, B, C}{\vdash \Gamma, \Delta, A \otimes B, C} \qquad \dfrac{\vdash_\xi \Gamma, A \qquad \vdash_\varphi \Delta, B, D}{\vdash \Gamma, \Delta, A \otimes B, D}}{\vdash \Gamma, \Delta, A \otimes B, C \,\&\, D} \longrightarrow \dfrac{\vdash_\xi \Gamma, A \qquad \dfrac{\vdash_\zeta \Delta, B, C \qquad \vdash_\varphi \Delta, B, D}{\vdash \Delta, B, C \,\&\, D}}{\vdash \Gamma, \Delta, A \otimes B, C \,\&\, D}$$

Unsurprisingly, the N/N and P/P permutations are freely allowed in MALL-S.

Definition 19 (equipollent permutation)
Suppose $\sum_{i \in I} \prod_{j \in J_i} \hat{E}_{ij} \Vdash E$ *and* $\sum_{k \in K} \prod_{l \in L_k} \hat{F}_{kl} \Vdash F$

1. *A P/P permutation (for any $u \in I$, $v \in J_u$, $w \in K$ and $x \in L_w$) is as follows:*

$$\dfrac{\left\langle \vdash_{\zeta(l)} \Gamma_l, \Downarrow\left(\hat{F}_{wl}\right)^+ \right\rangle_{l \in L_w \setminus x} \qquad \dfrac{\vdash_\varphi \Omega, \Downarrow\left(\hat{E}_{uv}\right)^+, \Downarrow\left(\hat{F}_{wx}\right)^+}{\vdash \Omega, \{\Gamma_l\}_{l \in L_w \setminus x}, (F)^+, \Downarrow\left(\hat{E}_{uv}\right)^+}\,\mathrm{P}}{\dfrac{\left\langle \vdash_{\xi(j)} \Delta_j, \Downarrow\left(\hat{E}_{uj}\right)^+ \right\rangle_{j \in J_u \setminus v}}{\vdash \Omega, \{\Delta_j\}_{j \in J_u \setminus v}, \{\Gamma_l\}_{l \in L_w \setminus x}, (E)^+, (F)^+}\,\mathrm{P}}$$

$$\dfrac{\left\langle \vdash_{\xi(l)} \Gamma_l, \Downarrow\left(\hat{F}_{wl}\right)^+ \right\rangle_{l\in L_w\setminus x} \quad \dfrac{\left\langle \vdash_{\xi(j)} \Delta_j, \Downarrow\left(\hat{E}_{uj}\right)^+ \right\rangle_{j\in J_u\setminus v} \quad \vdash_{\varphi} \Omega, \Downarrow\left(\hat{E}_{uv}\right)^+, \Downarrow\left(\hat{F}_{wx}\right)^+}{\vdash \Omega, \{\Delta_j\}_{j\in J_u\setminus v}, \Downarrow\left(\hat{F}_{wx}\right)^+, (E)^+}\,\mathrm{P}}{\vdash \Omega, \{\Delta_j\}_{j\in J_u\setminus v}, \{\Gamma_l\}_{l\in L_w\setminus x}, (E)^+, (F)^+}\,\mathrm{P}$$

$$\longrightarrow \quad \dfrac{\dfrac{\forall k \in K. \quad \vdash_{\xi(i,k)} \Delta, \left\{\Downarrow\left(\hat{F}_{kl}\right)^-\right\}_{l\in L_k}, \left\{\Downarrow\left(\hat{E}_{ij}\right)^-\right\}_{j\in J_i}}{\forall i \in I. \quad \vdash \Delta, (F)^-, \left\{\Downarrow\left(\hat{E}_{ij}\right)^-\right\}_{j\in J_i}}\,\mathrm{N}}{\vdash \Delta, (E)^-, (F)^-}\,\mathrm{N}$$

$$\dfrac{\dfrac{\forall i \in I. \quad \vdash_{\xi(i,k)} \Delta, \left\{\Downarrow\left(\hat{F}_{kl}\right)^-\right\}_{l\in L_k}, \left\{\Downarrow\left(\hat{E}_{ij}\right)^-\right\}_{j\in J_i}}{\forall k \in K. \quad \vdash \Delta, (E)^-, \left\{\Downarrow\left(\hat{F}_{kl}\right)^-\right\}_{l\in L_k}}\,\mathrm{N}}{\vdash \Delta, (E)^-, (F)^-}\,\mathrm{N}$$

Equipollent permutations have no restrictions on the form of the left of $\longrightarrow$, so these permutations are always allowed. Of the two remaining permutation forms, the N/P permutation is also always valid and readily defined.

Definition 20 (N/P permutation)
Suppose $\sum_{i\in I}\prod_{j\in J_i}\hat{E}_{ij} \Vdash E$ *and* $\sum_{k\in K}\prod_{l\in L_k}\hat{F}_{kl} \Vdash F$. *An N/P permutation is of the following form (for any* $u \in I$ *and* $v \in J_u$*)*

$$\dfrac{\left\langle \vdash_{\xi(j)} \Delta_j, \Downarrow\left(\hat{E}_{uj}\right)^+ \right\rangle_{j\in J_u\setminus v} \quad \dfrac{\forall k \in K. \quad \vdash_{\zeta(k)} \Gamma, \Downarrow\left(\hat{E}_{uv}\right)^+, \left\{\Downarrow\left(\hat{F}_{kl}\right)^-\right\}_{l\in L_k}}{\vdash \Gamma, (F)^-, \Downarrow\left(\hat{E}_{uv}\right)^+}\,\mathrm{N}}{\vdash \{\Delta\}_{j\in J_u}, \Gamma, (E)^+, (F)^-}\,\mathrm{P}$$

$$\longrightarrow \quad \dfrac{\dfrac{\left\langle \vdash_{\xi(j)} \Delta_j, \Downarrow\left(\hat{E}_{uj}\right)^+ \right\rangle_{j\in J_u\setminus v} \quad \vdash_{\zeta(k)} \Gamma, \Downarrow\left(\hat{E}_{uv}\right)^+, \left\{\Downarrow\left(\hat{F}_{kl}\right)^-\right\}_{l\in L_k}}{\forall k \in K. \quad \vdash \Gamma, \{\Delta\}_{j\in J_u}, (E)^+, \left\{\Downarrow\left(\hat{F}_{kl}\right)^-\right\}_{l\in L_k}}\,\mathrm{P}}{\vdash \{\Delta\}_{j\in J_u}, \Gamma, (E)^+, (F)^-}\,\mathrm{N}$$

The final permutations are the P/N permutations, which are not generally permissible. In fact, writing this permutation type as P/N is somewhat misleading because actually several *coherent* instances of P in the premises of the bottom N rule will be merged. Two P instances are coherent if, essentially, they make the same disjunctive and multiplicative choices. However, they cannot be exactly identical because they have different conclusions.

Definition 21 (P/N permutation)
Suppose $\sum_{i\in I}\prod_{j\in J_i}\hat{E}_{ij} \Vdash E$ *and* $\sum_{k\in K}\prod_{l\in L_k}\hat{F}_{kl} \Vdash F$. *A P/N permutation is of the following form (for any* $u \in K$ *and* $v \in L_u$*)*

$$\dfrac{\forall i \in I. \quad \dfrac{\left\langle \vdash_{\xi(l)} \Delta_l, \Downarrow\left(\hat{F}_{ul}\right)^+ \right\rangle_{l \in L_u \setminus v} \qquad \vdash_{\zeta(i)} \Gamma, \Downarrow\left(\hat{F}_{uv}\right)^+, \left\{\Downarrow\left(\hat{E}_{ij}\right)^-\right\}_{j \in J_i}}{\vdash \Gamma, \{\Delta_l\}_{l \in L_u}, \left\{\Downarrow\left(\hat{E}_{ij}\right)^-\right\}_{j \in J_i}, (F)^+}\ \mathrm{P}}{\vdash \Gamma, \{\Delta_l\}_{l \in L_u}, (E)^-, (F)^+}\ \mathrm{N}$$

$$\longrightarrow \quad \dfrac{\left\langle \vdash_{\xi(l)} \Delta_l, \Downarrow\left(\hat{F}_{ul}\right)^+ \right\rangle_{l \in L_u \setminus v} \qquad \dfrac{\forall i \in I. \quad \vdash_{\zeta(i)} \Gamma, \Downarrow\left(\hat{F}_{uv}\right)^+, \left\{\Downarrow\left(\hat{E}_{ij}\right)^-\right\}_{j \in J_i}}{\vdash \Gamma, \Downarrow\left(\hat{F}_{uv}\right)^+, (E)^-}\ \mathrm{N}}{\vdash \Gamma, \{\Delta_l\}_{l \in L_u}, (E)^-, (F)^+}\ \mathrm{P}$$

Observe how this permutation is restricted: all instances of P above the N must pick the same term in the sum pattern of $(F)^+$, must have all premises but one (the $\xi(l)$) exactly identical and independent of E, and the remaining premise (the $\zeta(i)$) in each case must contain all the subexpressions of E in that position in its sum pattern.

All the permutations defined are quite obviously sound (each application of P and N is correct), so we state the following lemma without proof.

Lemma 22. *The equipollent, P/N and N/P permutations are sound, i.e., if the left then the right hand side of* $\longrightarrow$. □

We end this section with a sketched proof that the N rule is invertible, using only synthetic permutations, instead of proving it in the usual way using cuts.

Theorem 23. *The N rule is invertible.*

Proof (Sketch). Since both N/N and N/P permutations are always allowed, every N rule can be permuted repeatedly towards the goal. Hence, for any derivation of $\vdash \Gamma, (E)^-$ that contains an instance of N for $(E)^-$, there is an equivalent derivation that begins with that N rule. If there are no instances of N in the proof of $\vdash \Gamma, (E)^-$, then there must be a subderivation that proves $\vdash \Gamma', (E)^-, (0)^-$ with N (because E is non-atomic). An instance of N for $(E)^-$ can be inserted here; each of its premises will contain $(0)^-$ and will therefore be provable. This instance of N can now be permuted to the goal. □

4 Strategies

In this section we shall assume that all polarisations are maximal (defn. 15).

As already seen, the MALL-S calculus, despite being a calculus of synthetic connectives, is more permissive in the order of synthetic rules than other focusing calculi such as LLF. However, LLF is recoverable in MALL-S as a *strategy* of applying inference rules to refine the goal sequent. By theorem. 23, the N rule is invertible, so it can always be applied to remove negatively polarised expressions from the context. Such propositions are only introduced to the context by the P rule. Therefore, ordinary focusing is a strategy of eagerly applying the N rules.

Definition 24 (ordinary focusing strategy). *The* focused *proofs of* $\vdash \Delta$ *are those that:*

1. *end in I; or*

2. *end with N if there are any negatively polarised propositions in Δ with the premises of the rule also focused; or*
3. *end with P if there are no negatively polarised propositions in Δ and all the premises of that rule are also focused.*

It is easy to see that this strategy degenerates to the familiar phase alternation after the pre-existing negatively polarised propositions in the goal sequent are removed, for each P step produces at-most one negatively polarised proposition in each premise, which upon decomposition produces only positively polarised premises. The completeness of this strategy is immediate from the invertibility of N (theorem. 23) and focalisation (cor. 18). We also state the following rather obvious instance of the cut-elimination algorithm (which is a synthetic restatement of the T-permutation [8]); we omit the proof.

Theorem 25 (focused cut-elimination). *The cut-elimination rewrite $\longrightarrow$ of theorem. 12 preserves focused proofs if the [NCC] case is given a higher precedence than the [PCC] case. That is, given focused proofs of $\vdash \Gamma, (A)^+$ and $\vdash \Delta, (A)^-$, the cut on A is eliminated to give a focused proof of $\vdash \Gamma, \Delta$.* □

As expected, this is not the sole interesting strategy for MALL-s proofs. In [5], a notion of *maximally multi-focused* proofs is introduced, which aims to equate all permutatively isomorphic MALL proofs in a unique syntax; such a proof exhibits the "true concurrency" inherent in the selection of foci. Maximality was defined there as a terminating permutative rewrite enlarging the principal formulas in a focusing calculus with multiple foci. The recipe in [5] for generating maximally multi-focused proofs from complete proofs can easily be repeated for MALL-s, but we do not pursue that direction here; instead, we characterise them here in terms of a strategy. In these maximal proofs the P rule rather than the N rule is eagerly applied, giving a tantalisingly dual picture from the strategy that generates ordinary LLF proofs.

Definition 26 (maximal multi-focusing strategy). *The* maximally multi-focused *(abbreviated as* maximal*) proofs of $\vdash \Delta$ are those proofs that:*

1. *end in I; or*
2. *end in P if there are any positively polarised propositions in Δ such that applying P (reading backwards) leads to a proof, and all the premises of this rule are also maximal; or*
3. *end in N if the situation for (2) does not apply, and the premises of this rule are also maximal.*

We state without the rather technical proof that the maximally multi-focused proofs in the sense of [5] are exactly those proofs in the above class up to equipollent permutations. Instead, we consider the question that was left open in [5] with regard to cut-elimination on maximal proofs.

Theorem 27 (maximal cut-elimination). *The cut-elimination rewrite $\longrightarrow$ in theorem. 12 preserves maximality if the [PCC] case is given a higher precedence than the [NCC] case. That is, given maximal proofs of $\vdash \Gamma, (A)^+$ and $\vdash \Delta, (A)^-$, the cut on A is eliminated to give a maximal proof of $\vdash \Gamma, \Delta$.*

Proof (Sketch). We reason by induction on the structure of the two input derivations for the cut being eliminated. The initial and principal cases are a straightforward application of the induction hypothesis. For the commutative cases, the result of the cut-elimination can only be maximal if the instances of P are kept closer to the root of the derivation, which requires prioritising a [PCC] rewrite over an [NCC] rewrite. □

The duality of theorems 25 and 27 is surprisingly clean, which gives further credence to the notion of maximally multi-focused proof. Of course, the above strategy is not implementable in a purely backwards reasoning (goal upwards) algorithm as it quantifies over proofs of the conclusion, which will not be available until the search completes. However, proofs in this class can be generated by saturation-based forward reasoning (axioms downwards) algorithms, such as in the inverse method [6]. Such search algorithms incrementally build a database of proved facts, so whenever a rule is applied resulting in a new fact, the N rules are permuted upwards in the proof as much as possible.

5 Conclusions and Future Work

We have presented a reconstruction of the calculus of synthetic connectives by means of polarisations of neutral proof patterns. Among the key technical merits of this presentation are simple and obviously correct proofs of cut, identity, focalisation, and synthetic permutations. We have shown how, among the proofs using synthetic inferences, the ordinary and maximally multi-focused proofs can be seen as diametrically opposite strategies, and demonstrated that the generic synthetic cut-elimination with a priority assignment preserves maximality.

The obvious next step is to extend the development to interesting fragments larger than MALL. We have already extended it to the exponentials and first-order quantifiers, but have left them out of this paper for presentational clarity and lack of space. Extending to infinite proof patterns (both sums and products) would also be interesting, requiring finite presentations of infinite operations; of particular interest is the question of proof patterns corresponding to the (co-)inductive connectives of μMALL [4]. Yet another important extension would be to a proof of synthetic cut-elimination for second-order MALL. Precise comparisons of synthetic derivations to game semantics would also be instructive.

References

1. Abramsky, S.: Sequentiality vs. concurrency in games and logic. Mathematical Structures in Computer Science 13(4), 531–565 (2003)
2. Abramsky, S., Melliès, P.-A.: Concurrent games and full completeness. In: 14th Symp. on Logic in Computer Science, pp. 431–442. IEEE Computer Society Press, Los Alamitos (1999)
3. Andreoli, J.-M.: Logic programming with focusing proofs in linear logic. J. of Logic and Computation 2(3), 297–347 (1992)
4. Baelde, D., Miller, D.: Least and greatest fixed points in linear logic. In: Dershowitz, N., Voronkov, A. (eds.) LPAR 2007. LNCS, vol. 4790, pp. 92–106. Springer, Heidelberg (2007)

5. Chaudhuri, K., Miller, D., Saurin, A.: Canonical sequent proofs via multi-focusing. In: Ausiello, G., Karhumäki, J., Mauri, G., Ong, L. (eds.) Fifth IFIP International Conference on Theoretical Computer Science. IFIP International Federation for Information Processing, vol. 273, pp. 383–396. Springer, Boston (2008)
6. Chaudhuri, K., Pfenning, F.: Focusing the inverse method for linear logic. In: Luke Ong, C.-H. (ed.) CSL 2005. LNCS, vol. 3634, pp. 200–215. Springer, Heidelberg (2005)
7. Chaudhuri, K., Pfenning, F., Price, G.: A logical characterization of forward and backward chaining in the inverse method. J. of Automated Reasoning 40(2-3), 133–177 (2008)
8. Danos, V., Joinet, J.-B., Schellinx, H.: LKT and LKQ: sequent calculi for second order logic based upon dual linear decompositions of classical implication. In: Girard, J.-Y., Lafont, Y., Regnier, L. (eds.) Advances in Linear Logic. London Mathematical Society Lecture Note Series, vol. 222, pp. 211–224. Cambridge University Press, Cambridge (1995)
9. Delande, O., Miller, D.: A neutral approach to proof and refutation in MALL. In: Pfenning, F. (ed.) 23th Symp. on Logic in Computer Science, pp. 498–508. IEEE Computer Society Press, Los Alamitos (2008)
10. Girard, J.-Y.: Locus solum. Mathematical Structures in Computer Science 11(3), 301–506 (2001)
11. Kleene, S.C.: Permutabilities of inferences in Gentzen's calculi LK and LJ. Memoirs of the American Mathematical Society 10, 1–26 (1952)
12. Laurent, O.: Etude de la polarisation en logique. Thèse de doctorat, Université Aix-Marseille II (March 2002)
13. Laurent, O.: Syntax vs. semantics: a polarized approach. Prépublication électronique PPS//03/04//no 17 (pp), Laboratoire Preuves, Programmes et Systèmes (Submitted) (March 2003)
14. Liang, C., Miller, D.: Focusing and polarization in linear, intuitionistic, and classical logics. Theoretical Computer Science (accepted, 2008)
15. Miller, D., Saurin, A.: From proofs to focused proofs: A modular proof of focalization in linear logic. In: Duparc, J., Henzinger, T.A. (eds.) CSL 2007. LNCS, vol. 4646, pp. 405–419. Springer, Heidelberg (2007)
16. Wadler, P.: Call-by-value is dual to call-by-name. In: 8th Int. Conf. on Functional Programming, pp. 189–201 (2003)
17. Zeilberger, N.: Focusing and higher-order abstract syntax. In: Necula, G.C., Wadler, P. (eds.) Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, pp. 359–369. ACM, New York (2008)
18. Zeilberger, N.: On the unity of duality. Ann. of Pure and Applied Logic (to appear, 2008)

An Algorithmic Interpretation of a Deep Inference System

Kai Brünnler and Richard McKinley

Institut für angewandte Mathematik und Informatik
Neubrückstr. 10, CH – 3012 Bern, Switzerland

Abstract. We set out to find something that corresponds to deep inference in the same way that the lambda-calculus corresponds to natural deduction. Starting from natural deduction for the conjunction-implication fragment of intuitionistic logic we design a corresponding deep inference system together with reduction rules on proofs that allow a fine-grained simulation of beta-reduction.

1 Introduction

The *Curry-Howard-Isomorphism* states that intuitionistic natural deduction derivations with the operation of detour-elimination behave exactly like lambda terms with beta reduction. For an introduction, see the book [4] by Girard, Lafont and Taylor. Since the lambda calculus expresses algorithms, the lambda calculus is thus an *algorithmic interpretation* of intuitionistic natural deduction. We want to find an algorithmic interpretation for *deep inference*, a formalism which has been introduced by Guglielmi [5]. So far, no deep inference system that we are aware of has an algorithmic interpretation. In fact, while they typically have cut elimination procedures, the cut elimination steps are not given in the form of simple reduction rules on proof terms.

The natural starting point for algorithmic interpretation of deep inference is of course a system for intuitionistic logic. There already exists such a system, by Tiu [11]. However, it focuses on locality of inference rules, and not on algorithmic interpretation. Its cut elimination proof works via translation to the sequent calculus. We design another deep inference system for intuitionistic logic, with the specific aim of staying as close to natural deduction as possible, because there the algorithmic interpretation is well-understood. We then give a definition of proof terms for this system. The general way of building proof terms for deep inference is already present in [1]. We equip these proof terms with reduction rules that allow us to simulate beta-reduction. We give translations from natural deduction to deep inference and back and prove a weak form of weak normalisation.

The principal way of composing our proof terms is not function application, as in the lambda calculus, but is function composition, as in composition of arrows in a category. So it is a system of what should be called *categorical combinators*. In fact, it turns out that our proof terms are very similar to some categorical

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 482–496, 2008.

combinators that Curien designed in the eighties, in order to serve as a target for the compilation of functional programming languages [2]. A very accessible introduction to those combinators and how they led to the development of explicit substitution calculi, like the $\lambda\sigma$-calculus, can be found in Hardin [7].

The difference between our combinators and Curien's is in the presentation of the defining adjunctions of a cartesian closed category. In our presentation proof terms can be thought of graphically: they are built using vertical composition (the usual composition of morphisms) and horizontal composition (the connectives).

2 A Deep Inference System for Intuitionistic Logic

Formulas, denoted by A, B, C, D, are defined as follows

$$A ::= a \mid (A \wedge A) \mid (A \supset A) \quad ,$$

where a is a propositional variable. As usual, conjunction binds stronger than implication and is left-associative, implication is right-associative. A *formula context*, denoted by $C\{\ \}$, is a formula with exactly one occurrence of the special propositional variable $\{\ \}$, called *the hole* or *the empty context*. The formula $C\{A\}$ is obtained by replacing the hole in $C\{\ \}$ by A. As usual, a context is *positive* if the number of implications we pass from the left on the path from the hole to the root is even. A context is *negative* if that number is odd, and is *strictly positive* if that number is zero.

A deep inference rule is a term rewriting rule on formulas. A rule is written

$$\rho \frac{A}{B} \quad ,$$

where ρ is the name of the rule and A and B are formulas containing schematic variables. A is the *premise* and B is the *conclusion* of the rule. In term rewriting A would be the left-hand-side or the redex and B would be the right-hand-side or the contractum. A *system* is a set of rules. An *instance* of a formula containing schematic variables is obtained by replacing the schematic variables by formulas. An *instance* of an inference rule as above is

$$\rho \frac{C\{A'\}}{C\{B'\}} \quad ,$$

where $C\{\ \}$ is a context, the formula A' is an instance of A and B' is an instance of B. A deep inference derivation is a sequence of rule instances composed in the obvious way. In term rewriting terminology a derivation is just a reduction sequence from one formula to another one using the given inference rules as rewrite rules. Of course, this definition only applies when the context is positive, since applying a rule in a negative context is generally unsound. For a negative context $C\{\ \}$, an instance of ρ will have the form

$$\rho \frac{C\{B'\}}{C\{A'\}} \quad .$$

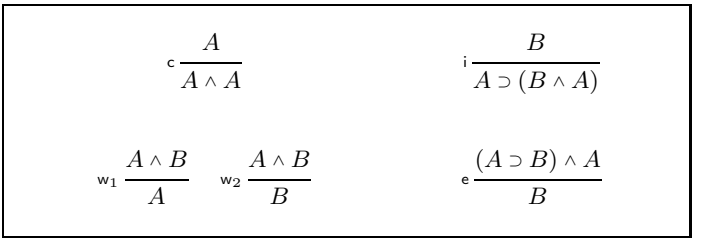


Fig. 1. A deep inference system for intuitionistic logic

Before seeing examples of derivations, let us look at a specific system for the conjunction-implication fragment of intuitionistic logic: the system in Figure 1. Because we like to think of the pair $\mathsf{w}_1, \mathsf{w}_2$ as one rule, there are essentially four rules: $\mathsf{c}, \mathsf{w}, \mathsf{i}, \mathsf{e}$, or, respectively: *contraction*, *weakening*, *implication introduction* and *implication elimination*. We can think of contraction as conjunction introduction and of weakening as conjunction elimination. Implication elimination can also be called *evaluation*. Categorically, each introduction rule is the unit and each elimination rule the counit of an adjunction. The system is designed with one goal in mind: to stay as close as possible to natural deduction, the home ground for algorithmic interpretation of proofs.

Let us now look at two examples of derivations. Notice how the inference rules apply deeply inside a context, as opposed to, say, rules in natural deduction. Notice also how the derivation on the right contains a "detour", a single instance of w_1 would also do the job.

Example 1

$$\dfrac{\dfrac{\dfrac{A \wedge B}{(A \wedge B) \wedge (A \wedge B)}\,\mathsf{c}}{B \wedge (A \wedge B)}\,\mathsf{w}_2}{B \wedge A}\,\mathsf{w}_1 \qquad \dfrac{\dfrac{\dfrac{A \wedge B}{(B \supset (A \wedge B)) \wedge B}\,\mathsf{i}}{(B \supset A) \wedge B}\,\mathsf{w}_1}{A}\,\mathsf{e}$$

We now introduce *proof terms*, or just *terms*, to capture deep inference derivations. Proof terms are denoted by R, T, U, V and are defined as follows:

$$R ::= \mathsf{id} \mid \rho \mid (R \,.\, R) \mid (R \wedge R) \mid (R \supset R)$$

where id is *identity*, ρ is the name of an inference rule from Figure 1, $(R_1 \,.\, R_2)$ is *(sequential) composition* and $(R_1 \wedge R_2)$ and $(R_1 \supset R_2)$ are *conjunction* and *implication*. Both conjunction and implication are also referred to as *parallel composition*. Sequential composition binds stronger than parallel composition and is left-associative. Unnecessary parentheses may be dropped.

Some proof terms can be typed. The typing judgement $A \xrightarrow{R} B$ says that the term R can have the type $A \to B$, so R has premise A and conclusion B. In that case R is called *typeable* and the triple consisting of A, R, B is called a *typed*

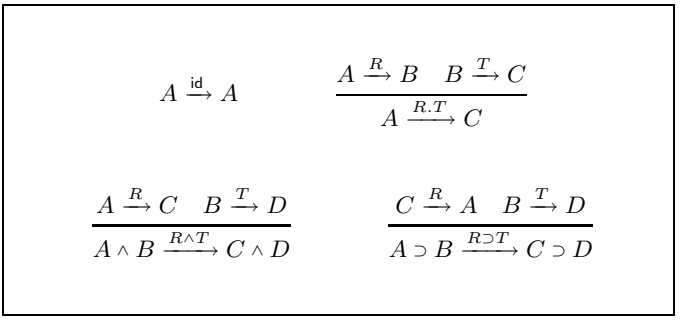


Fig. 2. Typing rules for proof terms

term. Typing judgements are derived by the typing rules in Figure 2 relative to a given set of typing axioms. A typing axiom types an inference rule name: we have $A \xrightarrow{\rho} B$ where A and B are instances of the premise and the conclusion of ρ, respectively. The only set of inference rules (or: typing axioms) we consider here is the one in Figure 1.

Example 2. Consider the following two terms R and T, which correspond to the derivations in Example 1:

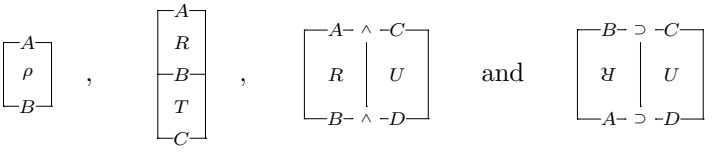
$$\mathsf{c} \,.\, (\mathsf{w}_2 \wedge \mathsf{id}) \,.\, (\mathsf{id} \wedge \mathsf{w}_1) \qquad \text{and} \qquad (\mathsf{i} \wedge \mathsf{id}) \,.\, ((\mathsf{id} \supset \mathsf{w}_1) \wedge \mathsf{id}) \,.\, \mathsf{e}$$

It is easy to see that they can be typed as $A \wedge B \xrightarrow{R} B \wedge A$ and $A \wedge B \xrightarrow{T} A$.

Clearly, there is canonical way of turning deep inference derivations into proof terms, as suggested by the examples above, and also a straightforward way of turning proof terms into deep inference derivations (that requires us to choose some order among parallel rewrites):

Proposition 1. *Given two formulas A, B and a system of inference rules, there is a derivation from A to B in that system iff there is a proof term R such that $A \xrightarrow{R} B$ can be derived from the typing axioms corresponding to the given system.*

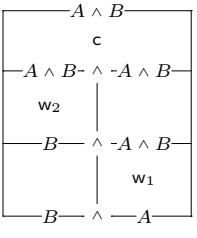
Having introduced these typing derivations, we replace them immediately by a more economical and suggestive notation, where we compose inference rules vertically and horizontally. Let ρ be an inference rule $A \xrightarrow{\rho} B$, and let $A \xrightarrow{R} B$, $B \xrightarrow{T} C$, $C \xrightarrow{U} D$. Then the typing derivations for ρ, $R \,.\, T$, $R \wedge U$ and $R \supset U$ are represented as the tiles



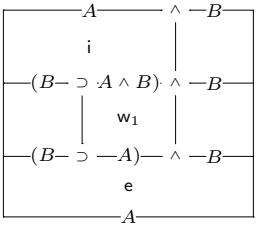
$$
\begin{array}{ll}
 & R \,.\, (T \,.\, U) \to (R \,.\, T) \,.\, U \\
 & R \,.\, \mathsf{id} \to R \leftarrow \mathsf{id} \,.\, R \\
(\mathsf{bur}) & \mathsf{id} \wedge \mathsf{id} \to \mathsf{id} \leftarrow \mathsf{id} \supset \mathsf{id} \\
 & (R \wedge T) \,.\, (U \wedge V) \to R \,.\, U \wedge T \,.\, V \\
 & (R \supset T) \,.\, (U \supset V) \to U \,.\, R \supset T \,.\, V \\
(\mathsf{nw}) & (R \wedge T) \,.\, \mathsf{w}_1 \to \mathsf{w}_1 \,.\, R \\
 & (R \wedge T) \,.\, \mathsf{w}_2 \to \mathsf{w}_2 \,.\, T \\
(\mathsf{nc}) & R \,.\, \mathsf{c} \to \mathsf{c} \,.\, (R \wedge R) \\
(\beta_\wedge) & \mathsf{c} \,.\, \mathsf{w}_1 \to \mathsf{id} \leftarrow \mathsf{c} \,.\, \mathsf{w}_2 \\
(\mathsf{ni}) & R \,.\, \mathsf{i} \to \mathsf{i} \,.\, (\mathsf{id} \supset (R \wedge \mathsf{id})) \\
(\beta_\supset) & (\mathsf{i} \,.\, (\mathsf{id} \supset R) \wedge T) \,.\, \mathsf{e} \to (\mathsf{id} \wedge T) \,.\, R
\end{array}
$$

Fig. 3. System beta

Example 3. Here are the tile representations of the derivations from the first example:



and



.

2.1 Reduction

Some reduction rules are shown in Figure 3. They were chosen for the single purpose of allowing us to simulate β-reduction of the simply-typed lambda calculus, the best-understood algorithmic interpretation of a logical system. In particular, the rules were not chosen to make sense categorically: some naturality equations are missing, extensionality is missing and the rule for beta reduction is more general than one would expect.

The system is called System beta. It has two subsystems that we wish to identify: System bur, the first block of reduction rules, which is labeled with (bur), and System subst, which is obtained from System beta by removing the $(\beta_\supset)$-rule. System bur equationally specifies a category with two bifunctors. From a deep inference point of view, it has nothing to do with the inference rules involved, it just equates derivations which differ due to inessential, bureaucratic detail. System subst is named in accordance with Curien. Consider a β-reduction step in the lambda calculus. There are two things to do: first, remove the application operator and the lambda, and second, carry out the substitution. While the $(\beta_\supset)$-rule allows us to do the first step, System subst allows us do the second step.

System beta is not locally confluent, its completion Beta is obtained by adding the rules in Figure 4. Morally, the right thing to do could be to work modulo

$$
\begin{array}{ll}
(\mathsf{bur}') & (W\,.\,(R \wedge T))\,.\,(U \wedge V) \to W\,.\,(R\,.\,U \wedge T\,.\,V) \\
 & (W\,.\,(R \supset T))\,.\,(U \supset V) \to W\,.\,(U\,.\,R \supset T\,.\,V) \\
(\mathsf{nw}') & (W\,.\,(R \wedge T))\,.\,\mathsf{w}_1 \to (W\,.\,\mathsf{w}_1)\,.\,R) \\
 & (W\,.\,(R \wedge T))\,.\,\mathsf{w}_2 \to (W\,.\,\mathsf{w}_2)\,.\,T) \\
 & (\mathsf{i} \wedge R)\,.\,\mathsf{e} \to \mathsf{id} \wedge R \\
(\beta_{\supset}') & (W\,.\,(\mathsf{i} \wedge R))\,.\,\mathsf{e} \to W\,.\,(\mathsf{id} \wedge R) \\
 & (W\,.\,((\mathsf{i}\,.\,(\mathsf{id} \supset R)) \wedge T))\,.\,\mathsf{e} \to (W\,.\,(\mathsf{id} \wedge T))\,.\,R
\end{array}
$$

Fig. 4. The completion of system beta into system Beta

bur, which would allow us to abandon these extra reduction rules. In this work we formally stay within the free theory. Nevertheless, we think of the terms as deep inference derivations, which are equal modulo associativity and, morally, should be equal modulo bur. System Bur is a completion of System bur and System Subst a completion of System subst, both are obtained by adding the corresponding rules from Figure 4.

For a given subsystem of System Beta we write $R \to T$ if R can be rewritten into T in one step by any rule in the given subsystem, so $\to$ is closed under context and irreflexive. We write $\to^n$ for the composition of $\to$ with itself n-times, and $\twoheadrightarrow$ for the reflexive-transitive closure of $\to$. If no subsystem is specified we mean System Beta itself.

Example 4. Our example terms R and T rewrite as follows:

$$
\begin{array}{ll}
 & \mathsf{c}\,.\,(\mathsf{w}_2 \wedge \mathsf{id})\,.\,(\mathsf{id} \wedge \mathsf{w}_1) \\
\to & \mathsf{c}\,.\,(\mathsf{w}_2\,.\,\mathsf{id}) \wedge (\mathsf{id}\,.\,\mathsf{w}_1) \\
\to^2 & \mathsf{c}\,.\,(\mathsf{w}_2 \wedge \mathsf{w}_1)
\end{array}
\qquad \text{and} \qquad
\begin{array}{ll}
 & (\mathsf{i} \wedge \mathsf{id})\,.\,((\mathsf{id} \supset \mathsf{w}_1) \wedge \mathsf{id})\,.\,\mathsf{e} \\
\to & (\mathsf{i}\,.\,(\mathsf{id} \supset \mathsf{w}_1) \wedge (\mathsf{id}\,.\,\mathsf{id}))\,.\,\mathsf{e} \\
\to & (\mathsf{id} \wedge \mathsf{id}\,.\,\mathsf{id})\,.\,\mathsf{w}_1 \\
\to^3 & \mathsf{w}_1
\end{array}
$$

Figure 5 shows for most of the reductions in system beta that they preserve typing. For the remaining rules this is easy to check, so we have the following proposition.

Proposition 2 (reduction preserves typing). *Let R and T be proof terms with $R \twoheadrightarrow T$. If $A \xrightarrow{R} B$ then $A \xrightarrow{T} B$.*

By checking critical pairs we get local confluence, strong normalisation for Bur can be obtained by a simple polynomial interpretation, so we have the following proposition.

Proposition 3
(i) Systems Bur, Subst *and* Beta *are locally confluent.*
(ii) System Bur *is confluent and strongly normalising.*

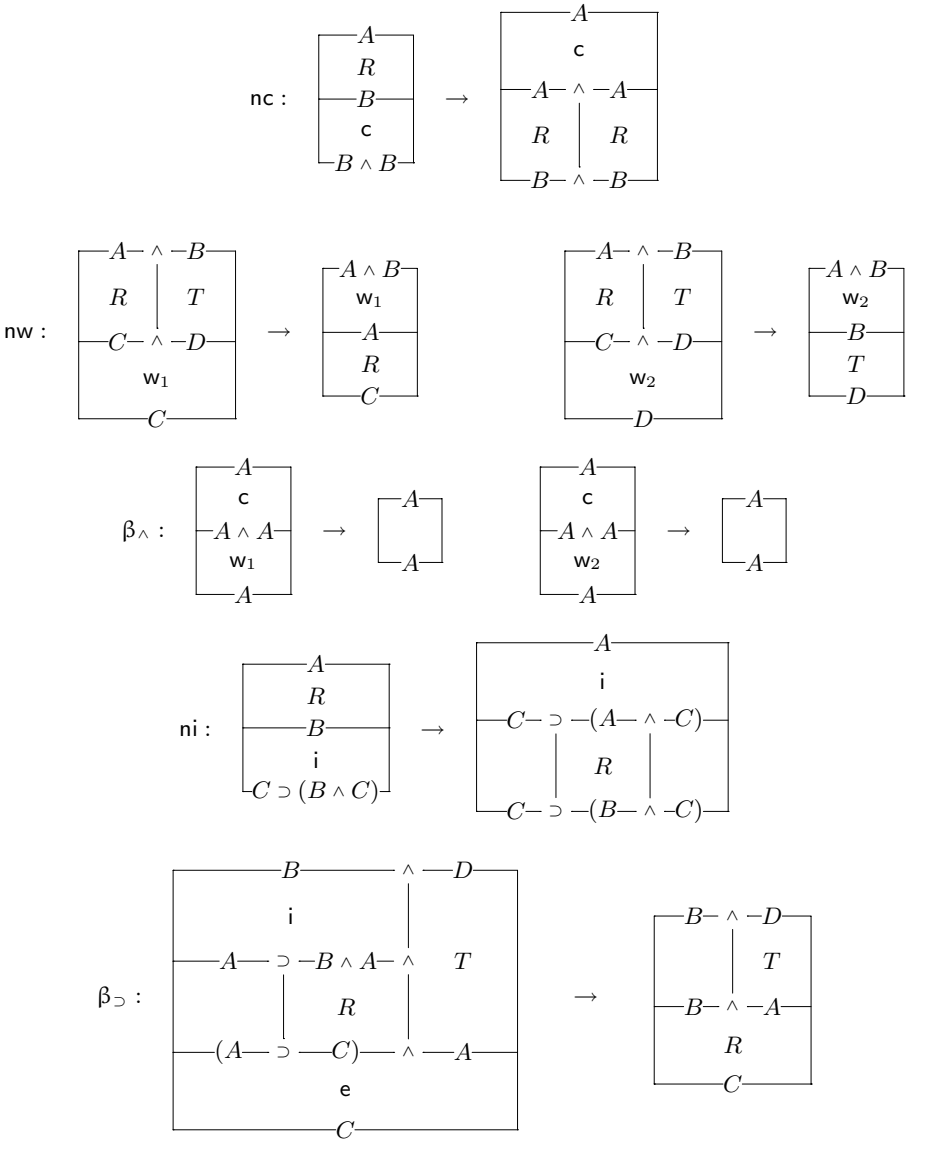


Fig. 5. Reduction rules with typing

Remark 1 System Subst, and thus also System Beta, is not strongly normalising. We have the following cycle:

$$\begin{aligned} \mathsf{c} \,.\, \mathsf{w}_1 \,.\, \mathsf{c} \,.\, \mathsf{w}_1 \quad &\to \quad \mathsf{c} \,.\, (\mathsf{c} \,.\, \mathsf{w}_1 \wedge \mathsf{c} \,.\, \mathsf{w}_1) \,.\, \mathsf{w}_1 \\ &\to \quad \mathsf{c} \,.\, \mathsf{w}_1 \,.\, (\mathsf{c} \,.\, \mathsf{w}_1) \\ &\to \quad \mathsf{c} \,.\, \mathsf{w}_1 \,.\, \mathsf{c} \,.\, \mathsf{w}_1 \qquad . \end{aligned}$$

The situation is different from Curien's system, where the subsystem for carrying out substitutions is strongly normalising. The confluence proofs for Curien's systems, that we know of, use strong normalisation of the subsystem which carries out substitutions, so they do not seem to directly apply in our setting. For the moment we do not know whether our system is confluent. In any case we do not see the failure of strong normalisation as a major defect. The problem is now to find a natural and liberal strategy which ensures termination.

3 The Relation with Natural Deduction

There is an obvious inductive translation of a natural deduction derivation into a deep inference derivation. It yields a deep inference derivation with the same conclusion as the natural deduction derivation and which has as its premise the conjunction of all premises of the natural deduction derivation. Since our inference rules are all sound and since a suitable replacement theorem holds for intuitionistic logic, we also know that we can also embed deep inference into natural deduction. So translations in both directions exist. However, they only work on derivations, not on the underlying untyped terms. What we would like to have in both directions is a translation of untyped terms which has the property of preserving typing. However, the obvious inductive translation of derivations is not even well-defined on their underlying untyped terms. Consider a standard sequent-style natural deduction system with additive context treatment and without structural rules. The two axiom instances $A \vdash A$ and $B, A \vdash A$ are different derivations, that should be translated into $A \xrightarrow{\mathsf{id}} A$ and $B \wedge A \xrightarrow{\mathsf{w}_2} A$, respectively. However, both axiom instances have the same underlying pure term, namely just a variable. Clearly, taking the underlying pure lambda term loses too much information of the original derivation. To keep that information we very slightly extend the syntax of lambda terms. We mark a variable if it corresponds to an axiom of the first kind and we will not mark it if it corresponds to an axiom of the second kind. The marked variables behave as usual except that they are not allowed to be bound.

We consider λ-terms with de Bruijn indices, introduced in [3]. They are defined as follows, where $n \geq 1$:

$$M ::= n \mid n^{\cdot} \mid (\lambda M) \mid (M\, M) \mid (\pi_1 M) \mid (\pi_2 M) \mid \langle M, M \rangle \qquad ,$$

and where in a given term an occurrence of $n^{\cdot}$, a *marked* index, is in the scope of at most $n-1$ λ's. The reduction rules for β-reduction together with substitution $M[n \leftarrow N]$ and lifting t_i^n are defined as follows:

$$\begin{array}{ll} \pi_1\langle M, N\rangle \to M & m[n \leftarrow N] = \begin{cases} m-1 & m > n \\ t_0^{n-1}(N) & m = n \\ m & m < n \end{cases} \\ \pi_2\langle M, N\rangle \to N & m^{\cdot}[n \leftarrow N] = (m-1)^{\cdot} \\ (\lambda M)N \to M[1 \leftarrow N] & (M_1\, M_2)[n \leftarrow N] = (M_1[n \leftarrow N]\, M_2[n \leftarrow N]) \\ & (\lambda M)[n \leftarrow N] = (\lambda M[n+1 \leftarrow N]) \end{array}$$

$$\Gamma, A, \Delta \vdash i^{(\cdot)} : A \quad \text{where } i = |A, \Delta| \text{ and } i \text{ is marked iff } |\Gamma| = 0$$

$$\wedge_I \frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash \langle M, N\rangle : A \wedge B} \qquad \supset_I \frac{\Gamma, A \vdash M : B}{\Gamma \vdash \lambda M : A \supset B}$$

$$\wedge_E \frac{\Gamma \vdash M : A \wedge B}{\Gamma \vdash \pi_1 M : A} \quad \wedge_E \frac{\Gamma \vdash M : A \wedge B}{\Gamma \vdash \pi_2 M : B} \qquad \supset_E \frac{\Gamma \vdash M : A \supset B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$

Fig. 6. Typing rules for the name-free λ-calculus

$$t_i^n(m) = \begin{cases} m+n & m > i \\ m & m \le i \end{cases} \qquad t_i^n(M\,N) = (t_i^n(M)\,t_i^n(N))$$
$$t_i^n(m^{\cdot}) = (m+n)^{\cdot} \qquad t_i^n(\lambda M) = (\lambda\, t_{i+1}^n(M))$$

A *typing context*, denoted by Γ or Δ, is a finite sequence of formulas. For typing context Γ its length is denoted by $|\Gamma|$ and the conjunction of all its formulas, in the given order and associated to the left, is denoted by $\wedge\Gamma$. Our typing system for lambda terms is given in Figure 6. Notice that it is impossible to type any term in an empty context, because that would require us to abstract over a marked index, which is not allowed. Let $\top$ denote $a \supset a$, for some atom a. Notice that whenever $\Gamma \vdash M : A$ and M' is obtained from M by removing all markings, then $\top, \Gamma \vdash M' : A$

Natural deduction to deep inference. We define a function $\underline{\quad}_{\mathsf{D}}$ from λ-terms to deep inference proof terms. We write R^n for $n > 0$ to denote R sequentially composed with itself n times. An expression $R^0 \,.\, T$ or $T \,.\, R^0$ denotes just T.

$$\underline{m^{\cdot}}_{\mathsf{D}} = \begin{cases} \mathsf{id} & m = 1 \\ \mathsf{w}_1^{m-1} & m > 1 \end{cases} \qquad \begin{aligned} \underline{\lambda M}_{\mathsf{D}} &= \mathsf{i} \,.\, (\mathsf{id} \supset \underline{M}_{\mathsf{D}}) \\ \underline{MN}_{\mathsf{D}} &= \mathsf{c} \,.\, (\underline{M}_{\mathsf{D}} \wedge \underline{N}_{\mathsf{D}}) \,.\, \mathsf{e} \\ \underline{\pi_n M}_{\mathsf{D}} &= \underline{M}_{\mathsf{D}} \,.\, \mathsf{w}_n \\ \underline{\langle M, N\rangle}_{\mathsf{D}} &= \mathsf{c} \,.\, (\underline{M}_{\mathsf{D}} \wedge \underline{N}_{\mathsf{D}}) \end{aligned}$$

$$\underline{m}_{\mathsf{D}} = \mathsf{w}_1^{m-1} \,.\, \mathsf{w}_2$$

It is straightforward to check that the embedding preserves typing, so we omit the proof, even though it is very instructive:

Theorem 1 (the embedding preserves typing). *If* $\Gamma \vdash M : A$ *then* $\wedge\Gamma \xrightarrow{\underline{M}_{\mathsf{D}}} A$.

We now come to the main theorem: System Beta can simulate β-reduction. The proof is of course similar to the proof of a similar result for Curien's combinators in [2]. We write $\mathsf{id}_n(R)$ for $(\ldots(R \wedge \underbrace{\mathsf{id}) \ldots \wedge \mathsf{id}}_{n \text{ times}})$.

Theorem 2 (the embedding preserves reduction)
(i) If $M \twoheadrightarrow_\beta N$ *then* $\underline{M}_{\mathsf{D}} \twoheadrightarrow \underline{N}_{\mathsf{D}}$.
(ii) $\mathsf{id}_{n-1}(\mathsf{c} \,.\, (\mathsf{id} \wedge \underline{N}_{\mathsf{D}})) \,.\, \underline{M}_{\mathsf{D}} \twoheadrightarrow \underline{M[n \leftarrow N]}_{\mathsf{D}}$
(iii) $\mathsf{id}_i(\mathsf{w}_1^n) \,.\, \underline{M}_{\mathsf{D}} \twoheadrightarrow \underline{t_i^n(M)}_{\mathsf{D}}$

Proof The first claim follows from the following diagram, which relies on (ii). A similar diagram works for the projection–pairing reduction.

$$\begin{array}{ccc}
(\lambda M)N & \xrightarrow{\quad -\!\!\!\circ_{\mathsf{D}} \quad} & \mathsf{c}\,.\,(\mathsf{i}\,.\,(\mathsf{id} \supset \underline{M}_{\mathsf{D}}) \wedge \underline{N}_{\mathsf{D}})\,.\,\mathsf{e} \\
\Big\downarrow{\scriptstyle \beta} & & \Big\downarrow{\scriptstyle \beta_{\supset}{}'} \\
 & & \mathsf{c}\,.\,(\mathsf{id} \wedge \underline{N}_{\mathsf{D}})\,.\,\underline{M}_{\mathsf{D}} \\
 & & \Big\downarrow{\scriptstyle (ii)} \\
M[1 \leftarrow N] & \xrightarrow{\quad -\!\!\!\circ_{\mathsf{D}} \quad} & \underline{M[1 \leftarrow N]}_{\mathsf{D}}
\end{array}$$

We now prove (ii), by induction on M. We see the cases for an index, an application, and an abstraction. The cases for a marked index, for pairing and for projection are straightforward.

$$\begin{aligned}
\mathsf{id}_{n-1}(\mathsf{c}\,.\,(\mathsf{id} \wedge \underline{N}_{\mathsf{D}}))\,.\,\underline{m}_{\mathsf{D}} \quad &= \quad \mathsf{id}_{n-1}(\mathsf{c}\,.\,(\mathsf{id} \wedge \underline{N}_{\mathsf{D}}))\,.\,\mathsf{w}_1^{m-1}\,.\,\mathsf{w}_2 \\
&\twoheadrightarrow \begin{cases}
\quad \mathsf{w}_1^{n-1}\,.\,\mathsf{c}\,.\,(\mathsf{id} \wedge \underline{N}_{\mathsf{D}})\,.\,\mathsf{w}_1^{m-n}\,.\,\mathsf{w}_2 & \\
\twoheadrightarrow \mathsf{w}_1^{m-2}\,.\,\mathsf{w}_2 = \underline{m-1}_{\mathsf{D}} = \underline{m[n \leftarrow N]}_{\mathsf{D}} & m > n \\
\quad \mathsf{w}_1^{n-1}\,.\,\mathsf{c}\,.\,(\mathsf{id} \wedge \underline{N}_{\mathsf{D}})\,.\,\mathsf{w}_2 & \\
\twoheadrightarrow \mathsf{w}_1^{n-1}\,.\,\underline{N}_{\mathsf{D}} \overset{(iii)}{\twoheadrightarrow} \underline{t_0^{n-1}(N)}_{\mathsf{D}} = \underline{m[n \leftarrow N]}_{\mathsf{D}} & m = n \\
\quad \mathsf{w}_1^{m-1}\,.\,\mathsf{id}_{n-m}(\mathsf{c}\,.\,(\mathsf{id} \wedge \underline{N}_{\mathsf{D}}))\,.\,\mathsf{w}_2 & \\
\twoheadrightarrow \mathsf{w}_1^{m-1}\,.\,\mathsf{w}_2 = \underline{m}_{\mathsf{D}} = \underline{m[n \leftarrow N]}_{\mathsf{D}} & m < n
\end{cases}
\end{aligned}$$

$$\begin{aligned}
\mathsf{id}_{n-1}(\mathsf{c}\,.\,(\mathsf{id} \wedge \underline{N}_{\mathsf{D}}))\,.\,\underline{M_1 M_2}_{\mathsf{D}} \quad &= \quad \mathsf{id}_{n-1}(\ldots)\,.\,\mathsf{c}\,.\,(\underline{M_1}_{\mathsf{D}} \wedge \underline{M_2}_{\mathsf{D}})\,.\,\mathsf{e} \\
&\twoheadrightarrow \quad \mathsf{c}\,.\,(\mathsf{id}_{n-1}(\ldots)\,.\,\underline{M_1}_{\mathsf{D}} \wedge \mathsf{id}_{n-1}(\ldots)\,.\,\underline{M_2}_{\mathsf{D}})\,.\,\mathsf{e} \\
&\twoheadrightarrow \quad \mathsf{c}\,.\,(\underline{M_1[n \leftarrow N]}_{\mathsf{D}} \wedge \underline{M_2[n \leftarrow N]}_{\mathsf{D}})\,.\,\mathsf{e} \\
&= \quad \underline{M_1[n \leftarrow N] M_2[n \leftarrow N]}_{\mathsf{D}} = \underline{(M_1 M_2)[n \leftarrow N]}_{\mathsf{D}}
\end{aligned}$$

$$\begin{aligned}
\mathsf{id}_{n-1}(\ldots)\,.\,\underline{\lambda M}_{\mathsf{D}} \quad &= \quad \mathsf{id}_{n-1}(\ldots)\,.\,\mathsf{i}\,.\,(\mathsf{id} \supset \underline{M}_{\mathsf{D}}) \\
&\twoheadrightarrow \quad \mathsf{i}\,.\,(\mathsf{id} \supset \mathsf{id}_n(\ldots))\,.\,(\mathsf{id} \supset \underline{M}_{\mathsf{D}}) \\
&\twoheadrightarrow \quad \mathsf{i}\,.\,(\mathsf{id} \supset \mathsf{id}_n(\ldots)\,.\,\underline{M}_{\mathsf{D}}) \\
&\twoheadrightarrow \quad \mathsf{i}\,.\,(\mathsf{id} \supset \underline{M[n+1 \leftarrow N}_{\mathsf{D}}) \\
&= \quad \underline{\lambda(M[n+1 \leftarrow N])}_{\mathsf{D}} = \underline{(\lambda M)[n \leftarrow N]}_{\mathsf{D}}
\end{aligned}$$

We now prove (iii), again by induction on M. We again see the cases for an index, an application and an abstraction, the cases for a marked index, a pairing

and a projection are straightforward.

$$\mathsf{id}_i(\mathsf{w}_1^n)\,.\,\underline{m}_{\mathsf{D}} \quad = \quad (\ldots(\mathsf{w}_1^n \wedge \underbrace{\mathsf{id}) \ldots \wedge \mathsf{id}}_{i \text{ times}})\,.\,\mathsf{w}_1^{m-1}\,.\,\mathsf{w}_2 \quad \twoheadrightarrow \quad \begin{cases} \mathsf{w}_1^{m-1}\,.\,\mathsf{w}_2 = \underline{m}_{\mathsf{D}} = \underline{t_i^n(m)}_{\mathsf{D}} & m \leq i \\ \mathsf{w}_1^{m-1+n}\,.\,\mathsf{w}_2 = \underline{m+n}_{\mathsf{D}} = \underline{t_i^n(m)}_{\mathsf{D}} & m > i \end{cases} .$$

$$\begin{aligned} \mathsf{id}_i(\mathsf{w}_1^n)\,.\,\underline{M_1 M_2}_{\mathsf{D}} \quad &= \quad \mathsf{id}_i(\mathsf{w}_1^n)\,.\,(\mathsf{c}\,.\,(\underline{M_1}_{\mathsf{D}} \wedge \underline{M_2}_{\mathsf{D}})\,.\,\mathsf{e}) \\ &\twoheadrightarrow \quad \mathsf{c}\,.\,(\mathsf{id}_i(\mathsf{w}_1^n)\,.\,\underline{M_1}_{\mathsf{D}} \wedge \mathsf{id}_i(\mathsf{w}_1^n)\,.\,\underline{M_2}_{\mathsf{D}})\,.\,\mathsf{e} \\ &\twoheadrightarrow \quad \mathsf{c}\,.\,(\underline{t_i^n M_1}_{\mathsf{D}} \wedge \underline{t_i^n M_2}_{\mathsf{D}})\,.\,\mathsf{e} \\ &= \quad \underline{t_i^n M_1}_{\mathsf{D}}\,\underline{t_i^n M_2}_{\mathsf{D}} = \underline{t_i^n(M_1 M_2)}_{\mathsf{D}} \quad . \end{aligned}$$

$$\begin{aligned} \mathsf{id}_i(\mathsf{w}_1^n)\,.\,\underline{\lambda N}_{\mathsf{D}} \quad &= \quad \mathsf{id}_i(\mathsf{w}_1^n)\,.\,(\mathsf{i}\,.\,(\mathsf{id} \supset \underline{N}_{\mathsf{D}})) \\ &\twoheadrightarrow \quad \mathsf{i}\,.\,(\mathsf{id} \supset \mathsf{id}_{i+1}(\mathsf{w}_1^n)\,.\,\underline{N}_{\mathsf{D}}) \\ &\twoheadrightarrow \quad \mathsf{i}\,.\,(\mathsf{id} \supset \underline{t_{i+1}^n N}_{\mathsf{D}}) = \underline{\lambda t_{i+1}^n N}_{\mathsf{D}} = \underline{t_i^n(\lambda N)}_{\mathsf{D}} \quad . \end{aligned}$$

□

Definition 1. *Let a proof term T be* essentially in normal form *if each reduction sequence in system* Beta *starting from T only contains instances of the rules $R\,.\,\mathsf{id} \to R$, $\mathsf{id}\,.\,R \to R$, $\mathsf{id} \wedge \mathsf{id} \to \mathsf{id}$ and $\mathsf{id} \supset \mathsf{id} \to \mathsf{id}$.*

Proposition 4 (the embedding essentially preserves normal form). *If M is in normal form then $\underline{M}_{\mathsf{D}}$ is essentially in normal form.*

Proof. By checking the reduction rules we first observe that, when given two terms R, T which are essentially in normal form, then also the terms $R \supset T$, $R \wedge T$, $\mathsf{i}\,.\,(R \supset T)$ and $\mathsf{c}\,.\,(R \wedge T)$ are essentially in normal form. We prove our proposition by induction on M. Translations of indices are clearly in normal form, and our observation takes care of abstractions and pairings, so we are left with applications and projections. Let M be an application $M_1 M_2$. Then M_1 can not be an abstraction, so it has to be either an index, a projection, a pairing or an application. Say it is an application $N_1 N_2$. Then $\underline{M}_{\mathsf{D}} = \mathsf{c}\,.\,(\underline{M_1}_{\mathsf{D}} \wedge \underline{M_2}_{\mathsf{D}})\,.\,\mathsf{e}$ with $\underline{M_1}_{\mathsf{D}} = \mathsf{c}\,.\,(\underline{N_1}_{\mathsf{D}} \wedge \underline{N_2}_{\mathsf{D}})\,.\,\mathsf{e}$. By induction hypothesis $\underline{M_1}_{\mathsf{D}}$ is essentially in normal form, so can only reduce to terms of the form $\mathsf{c}\,.\,U\,.\,\mathsf{e}$ or $\mathsf{c}\,.\,\mathsf{e}$. But then all reductions possible in a reduction sequence starting from $\underline{M}_{\mathsf{D}}$ are those that are either in a reduction sequence starting from $\underline{M_1}_{\mathsf{D}}$ or $\underline{M_2}_{\mathsf{D}}$ and thus $\underline{M}_{\mathsf{D}}$ is essentially in normal form. The other cases are similar.

Deep inference to natural deduction. We define a function $\_\__{\mathsf{N}}$ from deep inference proof terms to natural deduction proof terms, i.e. lambda terms. We give a definition using named lambda terms. For a given deep inference proof

term the function yields a lambda term with exactly one free variable, named x. The translation from that into a name-free lambda term is as usual, except that exactly those indices that come from occurrences of x are marked.

$$
\begin{array}{lll}
\underline{\mathsf{id}}_{\mathsf{N}} & = x & \\
\underline{\mathsf{w}_n}_{\mathsf{N}} & = \pi_n x & \underline{R \,.\, T}_{\mathsf{N}} = \underline{T}_{\mathsf{N}}[x \leftarrow \underline{R}_{\mathsf{N}}] \\
\underline{\mathsf{c}}_{\mathsf{N}} & = \langle x, x\rangle & \underline{R \wedge T}_{\mathsf{N}} = \langle \underline{R}_{\mathsf{N}}[x \leftarrow \pi_1 x], \underline{T}_{\mathsf{N}}[x \leftarrow \pi_2 x]\rangle \\
\underline{\mathsf{i}}_{\mathsf{N}} & = \lambda y.\langle x, y\rangle & \underline{R \supset T}_{\mathsf{N}} = \lambda y.\underline{T}_{\mathsf{N}}[x \leftarrow (x\underline{R}_{\mathsf{N}}[x \leftarrow y])] \quad (\text{fresh } y) \\
\underline{\mathsf{e}}_{\mathsf{N}} & = \pi_1 x \pi_2 x &
\end{array}
$$

Also the embedding in this direction preserves typing. Again it is straightforward to check and we have to omit the proof for space reasons.

Theorem 3 (the embedding preserves typing). *If* $A \xrightarrow{R} B$ *then* $A \vdash \underline{R}_{\mathsf{N}} : B$.

Remark 2. The embedding does not preserve normal form. Consider the normal form $\mathsf{i} \,.\, (\mathsf{id} \supset \mathsf{w}_1)$ which is mapped to $\lambda z.\pi_1(\lambda y\langle x, y\rangle z)$ which is not in normal form. The embedding does not preserve reduction. Consider the term $\mathsf{w}_1 \,.\, \mathsf{i}$ which reduces to $\mathsf{i} \,.\, (\mathsf{id} \supset (\mathsf{w}_1 \wedge \mathsf{id}))$ but $\underline{\mathsf{w}_1 \,.\, \mathsf{i}}_{\mathsf{N}} = \lambda y.\langle \pi_1 x, y\rangle$ is normal. The embedding does not preserve β-convertibility. Consider $\underline{\mathsf{id} \wedge \mathsf{id}}_{\mathsf{N}} = \langle \pi_1 x, \pi_2 x\rangle$ and $\underline{\mathsf{id}}_{\mathsf{N}} = x$. However, if $R \twoheadrightarrow T$ then $\underline{R}_{\mathsf{N}}$ and $\underline{T}_{\mathsf{N}}$ are convertible in lambda calculus with extensionality and surjective pairing.

Now we can use the two embeddings and their preservation of types to show the following theorem:

Theorem 4. *For each typed term there is a term in normal form with the same type.*

Proof. If a term R is typeable $A \xrightarrow{R} B$ then by Theorem 3 $A \vdash \underline{R}_{\mathsf{N}} : B$ and by weak normalisation and subject reduction of the typed lambda calculus $\underline{R}_{\mathsf{N}}$ has a normal form M with $A \vdash M : B$. Now $\underline{M}_{\mathsf{D}}$ is essentially normal by Proposition 4 and typeable $A \xrightarrow{M_{\mathsf{D}}} B$ by Theorem 1. Reducing $\underline{M}_{\mathsf{D}}$ in the canonical system formed by the four rules which collapse and remove identity we obtain a term T in normal form with $A \xrightarrow{T} B$.

Of course, while this is weak normalisation for *some* system, it is not weak normalisation for System Beta, since System Beta cannot simulate the effect of translating into the lambda calculus and back. So the problem now is to prove weak normalisation either directly or maybe by using a different embedding into lambda terms.

4 Discussion

Curien's combinators. We first explain the difference between our combinators and the categorical combinators of Curien. Both systems are orientations of a

subset of a defining set of equations of a cartesian closed category, see Lambek and Scott [8]. A cartesian closed category (without terminal object) is a category with binary products and exponentials, which correspond to conjunction and implication, respectively. Both of these structures may be defined using an *adjunction*. As explained in MacLane [9], an adjunction may be specified in many different ways, leading to different presentations of a cartesian closed category. Curien's system corresponds to the specification based on one functor, a mapping of arrows, and the counit. Our system corresponds to the more symmetric specification of an adjunction based on two functors and unit and counit. Curien's definition of a cartesian closed category is the one typically found in textbooks, such as [8].

The primitives for both systems are summarized in Figure 7. Each of the two rows represents an adjunction, and each column a collection of primitives. Our system takes the functors $\wedge$ and $\supset$ as primitive, while Curien takes the mappings $\langle -, - \rangle$ and Λ. For each adjunction we take both unit and counit, while Curien treats only the counit as a primitive. Of course, both systems include Δ implicitly. The terms of Curien's system are thus built from $\mathsf{id}, \mathsf{w}_1, \mathsf{w}_2, \mathsf{e}$ using arrow composition, and two constructors $\langle -, - \rangle$ and Λ. By the equivalence of the different presentations of an adjunction, we could define Curien's constructors as

$$\begin{array}{rcl} \Lambda(R) & = & A \xrightarrow{\mathsf{i}} B \supset (A \wedge B) \xrightarrow{\mathsf{id} \supset R} B \supset C \qquad \text{and} \\ \langle R, T \rangle & = & A \xrightarrow{\mathsf{c}} A \wedge A \xrightarrow{R \wedge T} B \wedge C \qquad . \end{array}$$

However, since we only have a subset of the defining equations of the adjunctions this will not lead to an embedding of Curien's system (not even the one called CCL_β since it contains a bit of surjective pairing). In particular Beta lacks naturality for the counit e, a part of naturality for the unit i as in $\mathsf{i} \,.\, (\mathsf{id} \supset (R \wedge T)) = \mathsf{i} \,.\, (T \supset (R \wedge \mathsf{id}))$, and the equations $\mathsf{c}.(\mathsf{w}_1 \wedge \mathsf{w}_2) = \mathsf{id}$ and $\mathsf{i}.(\mathsf{id} \supset \mathsf{e}) = \mathsf{id}$. Orienting and adding these equations would allow simulation of a lambda calculus with surjective pairing and extensionality and give equational equivalence with Curien's system $\mathsf{CCL}_{\beta\eta\mathsf{SP}}$.

Future work. Adding extensionality is an obvious route for further research. Adding full naturality for i and e is another interesting route: note that our embedding of the lambda calculus stays in the *strictly positive fragment* of proof terms, the fragment where the left-hand side of an implication is always the term id. System Beta never leaves the strictly positive fragment. Full naturality for i and e would allow us to leave that fragment. This gives us a lot of freedom. In explicit substitution calculi when a beta-redex is reduced a substitution arises from it, and then this substitution can be carried out indepently from *other* beta-redexes. In a system with full naturality a substitution could be carried out indepently even from *the very beta-redex that it arises from*. It would also be interesting to use the functor $\wedge$ to more economically embed lambda terms than what is possible with Curien's combinators: by distributing to each subterm not the entire environment, but only those variables of the environment that actually occur. This would correspond to embedding a natural deduction system with

left adjoint functor right adjoint functor	Hom-bijection	unit counit
$\Delta : f \mapsto (f, f)$ $\wedge : (f, g) \mapsto f \wedge g$	$\dfrac{(A, A) \xrightarrow{(f,g)} (B, C)}{A \xrightarrow{\langle f,g \rangle} B \wedge C}$	$\mathsf{c} : A \to A \wedge A$ $(\mathsf{w}_1, \mathsf{w}_2) :$ $(A \wedge B, A \wedge B) \to (A, B)$
$- \wedge A : f \mapsto f \wedge \mathsf{id}$ $A \supset - : f \mapsto \mathsf{id} \supset f$	$\dfrac{B \wedge A \xrightarrow{f} C}{B \xrightarrow{\Lambda(f)} A \supset C}$	$\mathsf{i} : B \to A \supset (B \wedge A)$ $\mathsf{e} : (A \supset B) \wedge A \to B$

Fig. 7. Primitives of both systems

multiplicative context treatment, and it would require some kind of exchange combinator, which shuffles around the channels corresponding to the variables. It would also be interesting to study *flow graphs* in the sense of [6] for our proof terms. It is easy to define them, and their acyclicity seems to be the key to a proof of normalisation. Our proof terms also give rise to interaction-style combinators, similar in spirit to those used for optimal reduction, but different because based on function composition instead of function application. An extension with more connectives would be interesting. Notice that the rules to add for disjunction are in perfect duality with those for conjunction:

$$\vee_I \frac{A}{A \vee B} \qquad \vee_I \frac{B}{A \vee B} \qquad \vee_E \frac{A \vee A}{A} \quad .$$

We enjoy this improvement over the situation in natural deduction, where we have essentially the same introduction rules, but the following elimination rule:

$$\frac{A \vee B \qquad \begin{array}{c}[A]\\ \vdots \\ C\end{array} \qquad \begin{array}{c}[B]\\ \vdots \\ C\end{array}}{C} \quad .$$

And finally, classical logic would be interesting. A sensible place would be to start with a system which can simulate reduction in the $\lambda\mu$-calculus [10].

References

1. Brünnler, K., Lengrand, S.: On two forms of bureaucracy in derivations. In: Bruscoli, P., Lamarche, F., Stewart, C. (eds.) Structures and Deduction, pp. 69–80. Technische Universität Dresden (2005)
2. Curien, P.-L.: Categorical Combinators, Sequential Algorithms and Functional Programming, 2nd edn. Research Notes in Theoretical Computer Science. Birkhäuser, Basel (1993)

3. de Bruijn, N.G.: Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. Indagationes Mathematicae (Proceedings) 75(5), 381–392 (1972)
4. Girard, J.-Y., Lafont, Y., Taylor, P.: Proofs and Types. Cambridge Tracts in Theoretical Computer Science, vol. 7. Cambridge University Press, Cambridge (1990)
5. Guglielmi, A.: A system of interaction and structure. ACM Transactions on Computational Logic 8(1), 1–64 (2007)
6. Guglielmi, A., Gundersen, T.: Normalisation control in deep inference via atomic flows. Logical Methods in Computer Science 4(1:9), 1–36 (2008), http://arxiv.org/pdf/0709.1205
7. Hardin, T.: From categorical combinators to $\lambda\sigma$-calculi, a quest for confluence. Technical report, INRIA Rocquencourt (1992), http://hal.inria.fr/inria-00077017/
8. Lambek, J., Scott, P.J.: Introduction to higher order categorical logic. Cambridge University Press, New York (1986)
9. Mac Lane, S.: Categories for the Working Mathematician. In: Graduate Texts in Mathematics. Springer, Heidelberg (1971)
10. Parigot, M.: $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction. In: Voronkov, A. (ed.) LPAR 1992. LNCS, vol. 624. Springer, Heidelberg (1992)
11. Tiu, A.F.: A local system for intuitionistic logic. In: Hermann, M., Voronkov, A. (eds.) LPAR 2006. LNCS, vol. 4246, pp. 242–256. Springer, Heidelberg (2006), http://users.rsise.anu.edu.au/~tiu/localint.pdf

Weak $\beta\eta$-Normalization and Normalization by Evaluation for System F

Andreas Abel

Department of Computer Science
Ludwig-Maximilians-University Munich

Abstract. A general version of the fundamental theorem for System F is presented which can be instantiated to obtain proofs of weak β- and $\beta\eta$-normalization and normalization by evaluation.

1 Introduction and Related Work

Dependently typed lambda-calculi have been successfully used as proof languages in proof assistants like Agda [19], Coq [16], LEGO [21], and NuPrl [11]. Since types may depend on values in these type theories, checking equality of types, which is crucial for type and, thus, proof checking, is non-trivial for these languages, and undecidable in the general case. In extensional type theories, such as the one underlying NuPrl, extensional, hence undecidable, type equality has been kept with the consequence that type checking is undecidable and requires user interaction. In intensional type theories, which are the basis of Agda, Coq, and LEGO, type equality has been restricted to a decidable fragment, called *definitional equality*, hence, type checking is decidable. However, the choice of this fragment strongly influences the comfort in using these systems: the more equal types are recognized as equal automatically, the fewer equality proofs the user has to construct manually.

Definitional equality encompasses at least computational equality, i. e., β, expanding of definitions, and recursion, and exactly like that it is currently implemented in Coq. However, there are suggestions to strengthen definitional equality by rewriting rules [8,10] and decision procedures [9]. On another line, it is strongly desirable to include η, which does not fit well under the rewriting paradigm. For incorporating η, normalization-by-evaluation has proven to be successful. Besides being explored for simple types [7,12,3] it has been extended to polymorphic types [4] and predicative dependent type theories [1]. Still open is its application to impredicative type theories such as the Calculus of Constructions (CoC), probably due to the difficult meta theory of CoC. Our long term goal is to formulate a verified normalization-by-evaluation (NbE) algorithm for impredicative type theories. This work is an important step towards this goal: we construct a generic model for System F whose instances are soundness and completeness proofs for NbE.

Altenkirch, Hofmann, and Streicher [4] already developed an NbE algorithm for System F and proved it correct. However, their work concerns only a combinatory (λ-free) version of System F. Moreover, they construct an internal model

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 497–511, 2008.

of System F in category theory; their work is only accessible to experts in categories, and the structure of the algorithm is a bit lost among the category-related technical details. They provide an SML-implementation of the algorithm in the appendix, but it is not formally related to the mathematical algorithm in the main text. In an unpublished article [5] they later extend their result to full System F (with λ-abstraction). Deep knowledge of category theory is a preliminary also for this paper, in the words of the authors, a "certain acquaintance with categories of presheaves" is assumed.

In this article, we try to give a more conventional presentation of NbE for System F, presuming only basic knowledge of λ-calculus and System F, domain theory, and inductive definitions in set theory. This way, we hope to make NbE for System F accessible to a wider audience, and to pave the way for an adaption of NbE to impredicative dependent type theories.

Overview. In Sec. 2, we introduce syntax and typing and computation rules for System F. A generic type interpretation is given in Sec. 3, and a generic formulation of the fundamental theorem for System F in Sec. 4. It is then instantiated to yield weak normalization proofs for β (Sec. 5) and $\beta\eta$ (Sec. 6). As main results we obtain soundness and completeness of an NbE algorithm for System F in Sec. 7.

2 Church-Style System F

In this section, we briefly recapitulate the syntax and static and dynamic semantics of System F. A more gentle introduction to System F can be found in Pierce's book [20, Ch. 23].

Syntax. Type variables X and term variables x are drawn from two distinct, countable supplies TyVar and Var.

$$\begin{array}{lll} \mathsf{Ty} \ni A, B, C & ::= X \mid A \to B \mid \forall X A & \text{types} \\ \mathsf{Tm} \ni r, s, t & ::= x \mid \lambda x{:}A.\, t \mid r\, s \mid \Lambda X t \mid r\, A & \text{terms} \\ \mathsf{Cxt} \ni \Gamma, \Delta & ::= \diamond \mid \Gamma, x{:}A & \text{typing contexts} \end{array}$$

We say context Γ' *extends* Γ, written $\Gamma' \leq \Gamma$, if $\Gamma'(x) = \Gamma(x)$ for all $x \in \mathsf{dom}(\Gamma)$. For instance, assuming $y \notin \mathsf{dom}(\Gamma)$, we have $(\Gamma, y{:}A) \leq \Gamma$, but not the other way round. Extending a context extends the set of terms typeable in that context; this theorem is called *weakening*.

Remark 1. The direction of $\leq$ is chosen to be compatible with subtyping. There, we let $\Gamma' \leq \Gamma$ if $\Gamma'(x) \leq \Gamma(x)$ for all $x \in \mathsf{dom}(\Gamma)$. Then $\Gamma \vdash t : A$, $\Gamma' \leq \Gamma$, and $A \leq A'$ imply $\Gamma' \vdash t : A'$ (contravariance!).

A substitution σ is a map from type variables to type expressions and from term variables to term expressions. We write $A\sigma$, $t\sigma$ for the simultaneous execution of substitution σ in A, t. As usual, $\mathsf{FV}(t)$ denotes the set of free type and term variables of t, and $\mathsf{FV}(A)$ the set of free type variables of A. Let $\mathsf{FV}(\Gamma) = \bigcup\{\mathsf{FV}(A) \mid (x{:}A) \in \Gamma\}$.

Typing (static semantics) $\Gamma \vdash t : A$.

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{\Gamma, x{:}A \vdash t : B}{\Gamma \vdash \lambda x{:}A.\, t : A \to B} \qquad \frac{\Gamma \vdash r : A \to B \qquad \Gamma \vdash s : A}{\Gamma \vdash r\, s : B}$$

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \Lambda X t : \forall X A}\ X \notin \mathsf{FV}(\Gamma) \qquad \frac{\Gamma \vdash t : \forall X A}{\Gamma \vdash t\, B : A[B/X]}$$

Operational (dynamic) semantics. One step $\beta\eta$-reduction is the closure of the following axioms under all term constructors:

$$\begin{array}{rlrl} (\lambda x{:}A.\, t)\, s & \longrightarrow_{\beta\eta} t[s/x] & \lambda x{:}A.\, t\, x & \longrightarrow_{\beta\eta} t \quad \text{if } x \notin \mathsf{FV}(t) \\ (\Lambda X t)\, A & \longrightarrow_{\beta\eta} t[A/X] & \Lambda X.\, t\, X & \longrightarrow_{\beta\eta} t \quad \text{if } X \notin \mathsf{FV}(t) \end{array}$$

We denote its reflexive-transitive closure by $\longrightarrow^*_{\beta\eta}$ and its reflexive-transitive-symmetric closure by $=_{\beta\eta}$.

In the metatheoretic investigation of System F to follow, we denote the dependent set-theoretic function space by $(x \in S) \to T(x)$, which is an abbreviation for $\{f \in S \to \bigcup_{x \in S} T(x) \mid f(x) \in T(x) \text{ for all } x \in S\}$.

3 Type Interpretation by Kripke Relations

We seek an interpretation of System F's types which is general enough to account for different normalization results. In particular, we are interested in normalization by evaluation, which exists in different flavors. To name a few, Berger and Schwichtenberg [7] interpret simple types as set-theoretical function spaces over the base type of term families, Filinski [14] uses continuous function spaces instead, and Altenkirch, Hofmann, and Streicher [4] construct a glueing model of System F types. We choose Abel, Coquand, and Dybjer's approach of *contextual reification* [2], where types are modelled as applicative structures with variables, and reification, i. e., converting semantic objects back to syntax, is context- and type-sensitive. This means that we need to interpret types as Kripke relations, i.e., relations indexed by contexts.

Kripke relations. Given a poset $(S, \subseteq)$, we say F is *Kripke* if $F \in \mathsf{Cxt} \to S$ and antitone, i.e., $\Gamma' \leq \Gamma$ implies $F(\Gamma) \subseteq F(\Gamma')$. We usually write F_Γ for $F(\Gamma)$.

We say D is an *applicative System F structure*, if D^A is Kripke for each $A \in \mathsf{Ty}$ and for all $A, B \in \mathsf{Ty}$, $X \in \mathsf{TyVar}$ and $\Gamma \in \mathsf{Cxt}$ there exist operations

$$\begin{array}{l} \mathsf{app}^{A,B}_\Gamma \in \mathsf{D}^{A \to B}_\Gamma \to \mathsf{D}^A_\Gamma \to \mathsf{D}^B_\Gamma, \\ \mathsf{App}^{X.A}_\Gamma \in \mathsf{D}^{\forall X A}_\Gamma \to (B \in \mathsf{Ty}) \to \mathsf{D}^{A[B/X]}_\Gamma. \end{array}$$

These operations need to be independent of type and context indices, i. e., $\mathsf{app}^{A,B}_\Gamma(f, d) = \mathsf{app}^{A',B'}_{\Gamma'}(f, d)$ if $f \in \mathsf{D}^{A \to B}_\Gamma \cap \mathsf{D}^{A' \to B'}_{\Gamma'}$ and $d \in \mathsf{D}^A_\Gamma \cap \mathsf{D}^{A'}_{\Gamma'}$, and similar for App. Thus, we can introduce an overloaded notation $\_\cdot\_$ for application

by $f \cdot d := \mathsf{app}(f, d)$ and $d \cdot B := \mathsf{App}(d, B)$. Examples for applicative System F structures are $\mathsf{Tm}^A_\Gamma := \{t \mid \Gamma \vdash t : A\}$, as well as $\mathsf{Tm}^A_\Gamma/{=_{\beta\eta}}$.

Let $\mathsf{D}, \hat{\mathsf{D}}$ be applicative System F structures. We define the set of *Kripke relations* of type A by

$$\mathcal{A} \in \mathbb{K}^A \iff \mathcal{A}_\Gamma \subseteq \mathsf{D}^A_\Gamma \times \hat{\mathsf{D}}^A_\Gamma \text{ for all } \Gamma \text{ and } \mathcal{A} \text{ is Kripke.}$$

We use the letters $\mathcal{A}, \mathcal{B}, \mathcal{C}$ for elements of $\mathbb{K}^A$. We write $\Gamma \vdash d \sim d' \in \mathcal{A}$ for $(d, d') \in \mathcal{A}_\Gamma$. $\mathbb{K}^A$ forms a complete lattice with $\subseteq$, $\cap$, and $\cup$ defined pointwise.

A Kripke relation $\mathcal{A}$ is a *Kripke PER* (partial equivalence relation) if $\mathcal{A}_\Gamma$ is symmetric and transitive for any context Γ, i.e., $\Gamma \vdash d \sim d' \in \mathcal{A}$ implies $\Gamma \vdash d' \sim d \in \mathcal{A}$ and $\Gamma \vdash d_1 \sim d_2 \in \mathcal{A}$ and $\Gamma \vdash d_2 \sim d_3 \in \mathcal{A}$ imply $\Gamma \vdash d_1 \sim d_3 \in \mathcal{A}$.

Constructing Kripke relations. In predicative type theories, such as Martin-Löf Type Theory, one can construct semantic types inductively, i. e., from below [1], without reference to syntax. For impredicative systems, like System F or the Calculus of Constructions, this is not possible. Instead, for each type constructor one has to define a matching operation in the interpretation domain of types, in our case, $\mathbb{K}$, and then define the interpretation of a type by *induction on syntax*, e. g., the size of the type expression [15, Sec. 14.2] or its derivation of wellfoundedness [22]. In the following, we provide the necessary constructions to interpret function type and universal type.

Kripke relations are closed under arbitrary intersections. Further constructions on Kripke relations are function space and type abstraction:

$$\begin{aligned}
_ \to _ \quad &\in \mathbb{K}^A \to \mathbb{K}^B \to \mathbb{K}^{A \to B} \\
(\mathcal{A} \to \mathcal{B})_\Gamma &= \{(f, f') \in \mathsf{D}^{A \to B}_\Gamma \times \hat{\mathsf{D}}^{A \to B}_\Gamma \mid \text{ for all } d, d', \Gamma' \le \Gamma, \\
&\qquad \Gamma' \vdash d \sim d' \in \mathcal{A} \text{ implies } \Gamma' \vdash f \cdot d \sim f' \cdot d' \in \mathcal{B}\}
\end{aligned}$$

$$\begin{aligned}
(_._)^{X.B} \quad &\in (A \in \mathsf{Ty}) \to \mathbb{K}^{B[A/X]} \to \mathbb{K}^{\forall XB} \\
(A.\mathcal{B})^{X.B}_\Gamma \quad &= \{(d, d') \in \mathsf{D}^{\forall XA}_\Gamma \times \hat{\mathsf{D}}^{\forall XA}_\Gamma \mid \Gamma \vdash d \cdot A \sim d' \cdot A \in \mathcal{B}\}
\end{aligned}$$

The function space $\mathcal{A} \to \mathcal{B}$ is monotone (covariant) in $\mathcal{B}$ and antitone (contravariant) in $\mathcal{A}$. The type abstraction operator $(A.\mathcal{B})^{X.B}$ is monotone in $\mathcal{B}$. In the following, we drop the superscript $X.B$.

Lemma 1. *If $\mathcal{A}, \mathcal{B}$ are Kripke PERs, so are $\mathcal{A} \to \mathcal{B}$ and $A.\mathcal{B}$.*

Interpretation space. Depending on what result one wants to harvest from a model construction for System F, one has to impose restrictions on the interpretation domain of types. For example, in a Tait-style proof of strong normalization using saturated sets, one requires each semantic type to be below the set $\mathcal{S}$ of strongly normalizing terms and above the set $\mathcal{N}$ of neutral strongly normalizing terms. Vaux [23] found an abstraction of $(\mathcal{N}, \mathcal{S})$ which he called *stable pair*. In the following, we present a further generalization which allows the restriction to be dependent on a syntactical type.

Definition 1 (Interpretation space). *An interpretation space consists of two Kripke relations $\underline{A} \subseteq \overline{A} \in \mathbb{K}^A$ for each type A such that the following conditions hold.*

K-FUN-E	$\underline{A \to B}$	$\subseteq \overline{A} \to \underline{B}$	
K-FUN-I	$\underline{A} \to \overline{B}$	$\subseteq \overline{A \to B}$	
K-ALL-E	$\underline{\forall Y\, A}$	$\subseteq B.\underline{A[B/Y]}$	*for any* B
K-ALL-I	$X.\overline{A[X/Y]}$	$\subseteq \overline{\forall Y\, A}$	*for a new* X

We write $A \Vdash \mathcal{A}$ (pronounced A realizes *$\mathcal{A}$) if $\underline{A} \subseteq \mathcal{A} \subseteq \overline{A}$.*

A trivial, not type-sensitive interpretation space is $\underline{A}_\Gamma = \mathcal{N}$ and $\overline{A}_\Gamma = \mathcal{S}$. In this case, $A \Vdash \mathcal{A}$ just means $\mathcal{A}$ is saturated. An analogy of $A \Vdash \mathcal{A}$ can be found in Matthes' proof of strong normalization of System F [18, Sec. 9.1.2] where it means $\mathcal{A}$ *is A-saturated*. More examples of interpretation spaces can be found in this article.

In the following, we assume an interpretation space. We now introduce the last construction on semantic types, quantification, which is relative to an interpretation space. If $\mathcal{F}(B) \in \mathbb{K}^B \to \mathbb{K}^{A[B/X]}$ for all $B \in \mathsf{Ty}$, we let

$$\bigcap \mathcal{F} = \bigcap \{B.\mathcal{F}(B, \mathcal{B}) \mid B \Vdash \mathcal{B}\} \in \mathbb{K}^{\forall X A}.$$

Intersection is restricted to realizable semantic types. This has an analogue in other normalization proofs of System F, e. g., Girard [15, Ch. 14] restricts quantification to reducibility candidates. The type constructions preserve realizability, thanks to the conditions imposed by Def. 1.

Lemma 2 (Realizability of type constructions)

1. *If $A \Vdash \mathcal{A}$ and $B \Vdash \mathcal{B}$ then $A \to B \Vdash \mathcal{A} \to \mathcal{B}$.*
2. *If $A[B/Y] \Vdash \mathcal{F}(B, \mathcal{B})$ for all $B \Vdash \mathcal{B}$, then $\forall Y\, A \Vdash \bigcap \mathcal{F}$.*

Proof Directly, using the postulates on $\underline{C}$ and $\overline{C}$.

1. First, we have $\underline{A \to B} \subseteq \overline{A} \to \underline{B}$ by K-FUN-E, and since $\mathcal{A} \subseteq \overline{A}$ and $\underline{B} \subseteq \mathcal{B}$ we obtain by contravariance of the function space $\underline{A \to B} \subseteq \mathcal{A} \to \mathcal{B}$.
   Secondly, since $\underline{A} \subseteq \mathcal{A}$ and $\mathcal{B} \subseteq \overline{B}$, contravariance yields $\mathcal{A} \to \mathcal{B} \subseteq \underline{A} \to \overline{B} \subseteq \overline{A \to B}$ by K-FUN-I.
2. First, for any $B \Vdash \mathcal{B}$ we have $\underline{A[B/Y]} \subseteq \mathcal{F}(B, \mathcal{B})$, thus by monotonicity of the abstraction operator, $B.\underline{A[B/Y]} \subseteq B.\mathcal{F}(B, \mathcal{B})$. By K-ALL-E it follows $\underline{\forall Y\, A} \subseteq B.\mathcal{F}(B, \mathcal{B})$, and since $B, \mathcal{B}$ were arbitrary, $\underline{\forall Y\, A} \subseteq \bigcap \mathcal{F}$.
   Secondly, let X be a new type variable. We have $X \Vdash \underline{X}$, hence, $A[X/Y] \Vdash \mathcal{F}(\underline{X}, \underline{X})$ which implies $\mathcal{F}(X, \underline{X}) \subseteq \overline{A[X/Y]}$. Thus, $\bigcap \mathcal{F} \subseteq X.\mathcal{F}(X, \underline{X}) \subseteq X.\overline{A[X/Y]} \subseteq \overline{\forall Y\, A}$ by K-ALL-I.

Type interpretation can now be defined mechanically, mapping the syntactic type constructors to the semantic ones. Let σ be a syntactical type substitution and $\rho(X) \in \mathbb{K}^{\sigma(X)}$ for all type variables X, which we write $\rho \in \mathbb{K}^{\sigma}$. We define $[\![A]\!]_\rho \in \mathbb{K}^{A\sigma}$ by the following equations.

$$\begin{aligned} [\![X]\!]_\rho &= \rho(X) \\ [\![A \to B]\!]_\rho &= [\![A]\!]_\rho \to [\![B]\!]_\rho \\ [\![\forall X A]\!]_\rho &= \bigcap((B \in \mathsf{Ty}) \mapsto (\mathcal{B} \in \mathbb{K}^B) \mapsto [\![A]\!]_{\rho[X \mapsto \mathcal{B}]}) \end{aligned}$$

Note that $\rho[X \mapsto \mathcal{B}] \in \mathbb{K}^{\sigma[X \mapsto B]}$ in the last line.

Lemma 3. *If $\rho(X)$ is a Kripke PER for all X, so is $[\![A]\!]_\rho$.*

Lemma 4 (Substitution). $[\![A[B/X]]\!]_\rho = [\![A]\!]_{\rho[X \mapsto [\![B]\!]_\rho]}$.

If $\rho \in \mathbb{K}^{\sigma}$, we let $\sigma \Vdash \rho$ if $\sigma(X) \Vdash \rho(X)$ for all type variables X. It is now easy to show that types realize their own interpretations.

Theorem 1 (Type interpretation is realizable). *If $\sigma \Vdash \rho$ then $A\sigma \Vdash [\![A]\!]_\rho$.*

4 Fundamental Lemma

A general model of System F can be given by interpreting terms in an applicative System F structure D and types as PERs over D. The fundamental theorem does not rely on types being PERs, thus, we can easily take Kripke relations instead. This freedom will be exploited for the soundness of NbE in Sec. 7.4.

Fix some context Δ. Let η map type variables to syntactical types and term variables to elements of D such that $\eta \in \mathsf{D}^{\Gamma}_{\Delta}$, meaning $\eta(x) \in \mathsf{D}^{B}_{\Delta}$ for all $(x\!:\!B) \in \Gamma$. Let $t \in \mathsf{Tm}^{A}_{\Gamma}$. We stipulate the existence of an evaluation function $(\!|t|\!)_\eta \in \mathsf{D}^{A}_{\Delta}$ with the following properties.

$$\begin{array}{lrl} \textsc{den-var} & (\!|x|\!)_\eta = \eta(x) & \\ \textsc{den-fun-e} & (\!|r\ s|\!)_\eta = (\!|r|\!)_\eta \cdot (\!|s|\!)_\eta & \\ \textsc{den-all-e} & (\!|r\ A|\!)_\eta = (\!|r|\!)_\eta \cdot A\eta & \\ \textsc{den-fun-i} & (\!|\lambda x\!:\!A.\, t|\!)_\eta \cdot d = (\!|t|\!)_{\eta[x \mapsto d]} & \text{if } d \in \mathsf{D}^{A}_{\Delta} \\ \textsc{den-all-i} & (\!|\Lambda X t|\!)_\eta \cdot A = (\!|t|\!)_{\eta[X \mapsto A]} & \end{array}$$

We call $(\mathsf{D}, \_\cdot\_, (\!|\_|\!)_\_)$ a *syntactical applicative System F structure* (cf. Barendregt [6, 5.3.1]).

Now we are ready to show the fundamental theorem. Let $(\hat{\mathsf{D}}, \_\cdot\_, (\!|\_|\!)'_\_)$ be another syntactical applicative System F structure. For $\eta \in \mathsf{D}^{\Gamma}_{\Delta}, \eta' \in \hat{\mathsf{D}}^{\Gamma}_{\Delta}$ we define

$$\Delta \vdash \eta \sim \eta' \in [\![\Gamma]\!]_\rho \iff \Delta \vdash \eta(x) \sim \eta'(x) \in [\![\Gamma(x)]\!]_\rho \text{ for all } x \in \mathsf{dom}(\Gamma).$$

Theorem 2 (Validity of typing). *Let $\eta \Vdash \rho$ and both $\eta \upharpoonright \mathsf{TyVar} = \eta' \upharpoonright \mathsf{TyVar}$ and $\Delta \vdash \eta \sim \eta' \in [\![\Gamma]\!]_\rho$. If $\Gamma \vdash t : A$ then $\Delta \vdash (\!|t|\!)_\eta \sim (\!|t|\!)'_{\eta'} \in [\![A]\!]_\rho$.*

Proof. By induction on $\Gamma \vdash t : A$. Interesting are the System F specific cases.

Case

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \Lambda X t : \forall X A}\ X \notin \mathsf{FV}(\Gamma)$$

$B \Vdash \mathcal{B}$	assumption
$\eta[X \mapsto B] \Vdash \rho[X \mapsto \mathcal{B}] =: \rho'$	by def.
$\Delta \vdash \eta[X \mapsto B] \sim \eta'[X \mapsto B] \in [\![\Gamma]\!]_\rho = [\![\Gamma]\!]_{\rho'}$	since $X \notin \mathsf{FV}(\Gamma)$
$\Delta \vdash (\!|t|\!)_{\eta[X\mapsto B]} \sim (\!|t|\!)'_{\eta'[X\mapsto B]} \in [\![A]\!]_{\rho'}$	by ind. hyp.
$\Delta \vdash (\!|\Lambda X t|\!)_\eta \cdot B \sim (\!|\Lambda X t|\!)'_{\eta'} \cdot B \in [\![A]\!]_{\rho'}$	DEN-ALL-I
$\Delta \vdash (\!|\Lambda X t|\!)_\eta \sim (\!|\Lambda X t|\!)'_{\eta'} \in B.[\![A]\!]_{\rho'}$	by def.
$\Delta \vdash (\!|\Lambda X t|\!)_\eta \sim (\!|\Lambda X t|\!)'_{\eta'} \in [\![\forall X A]\!]_\rho$	since $B \Vdash \mathcal{B}$ arbitrary

Case

$$\frac{\Gamma \vdash t : \forall X A}{\Gamma \vdash t\,B : A[B/X]}$$

$\Delta \vdash (\!|t|\!)_\eta \sim (\!|t|\!)'_{\eta'} \in \bigcap (B \mapsto \mathcal{B} \mapsto [\![A]\!]_{\rho[X\mapsto\mathcal{B}]})$	by ind.hyp.
$B\eta \Vdash [\![B]\!]_\rho$	by Thm. 1
$\Delta \vdash (\!|t|\!)_\eta \cdot B\eta \sim (\!|t|\!)'_{\eta'} \cdot B\eta \in [\![A]\!]_{\rho[X\mapsto[\![B]\!]_\rho]}$	by instantiation
$\Delta \vdash (\!|t\,B|\!)_\eta \sim (\!|t\,B|\!)'_{\eta'} \in [\![A]\!]_{\rho[X\mapsto[\![B]\!]_\rho]}$	DEN-ALL-E
$\Delta \vdash (\!|t\,B|\!)_\eta \sim (\!|t\,B|\!)'_{\eta'} \in [\![A[B/X]]\!]_\rho$	Substitution

5 Weak β-Normalization

From our general fundamental theorem we can recover a proof of β-normalization for System F. In this case, we construct an untyped interpretation space of discrete Kripke relations which ignores types and contexts. The following development is standard, we show here only that it fits into the abstractions we have chosen in sections 3 and 4.

Let $\overline{r}$ denote the β-equivalence class of r. Let $\mathsf{D}^A_\Gamma = \hat{\mathsf{D}}^A_\Gamma = \mathsf{Tm}/{=_\beta}$ for all Γ, A, with application defined by $\overline{r} \cdot \overline{s} = \overline{r\,s}$ and $\overline{r} \cdot A = \overline{r\,A}$. Evaluation is defined by $(\!|t|\!)_{\overline{\sigma}} = \overline{t\sigma}$.

Lemma 5. *Application and evaluation are well-defined and* $(\mathsf{D}, \_\cdot\_, (\!|\_|\!)\_)$ *forms a syntactical applicative System F structure.*

Neutral terms are given by the grammar $n ::= x \mid n\,s \mid n\,A$. The interpretation space for β-normalization is untyped, i. e., we set

$$\overline{A}_\Gamma = \mathcal{W}_\Gamma := \{(\overline{t}, \overline{t}) \mid t \text{ has a } \beta\text{-normal form } \}$$
$$\underline{A}_\Gamma = \mathcal{N}_\Gamma := \{(\overline{n}, \overline{n}) \mid n \text{ has a } \beta\text{-normal form } \}$$

for all types A. To show that these settings really constitute an interpretation space is a standard exercise.

Let η_{id} be the identity map on term and type variables, and let $\rho_{\mathsf{id}}(X) = \mathcal{N}$ for all $X \in \mathsf{TyVar}$. Clearly, $\eta_{\mathsf{id}} \Vdash \rho_{\mathsf{id}}$.

Lemma 6 (Identity environment). $\Gamma \vdash \overline{\eta_{\mathsf{id}}} \sim \overline{\eta_{\mathsf{id}}} \in [\![\Gamma]\!]_{\rho_{\mathsf{id}}}$.

Theorem 3 (Weak β-normalization of System F). *If $\Gamma \vdash t : A$ then t has a β-normal form.*

Proof. By the fundamental theorem $\Gamma \vdash (\!|t|\!)_{\overline{\eta_{\mathsf{id}}}} \sim (\!|t|\!)_{\overline{\eta_{\mathsf{id}}}} \in [\![A]\!]_{\rho_{\mathsf{id}}}$ which implies $\Gamma \vdash \overline{t} \sim \overline{t} \in \mathcal{W}$, hence, t has a β-normal form.

6 Weak $\beta\eta$-Normalization

In this section, we instantiate Thm. 2 to prove weak $\beta\eta$-normalization for System F. In particular, we show that each well-typed term has a η-long β-normal form. In this, we will require the Kripke aspect of our type interpretation, since *being η-long* for open terms can only be defined in the presence of a typing context.

Let now $\overline{r}$ denote the $\beta\eta$-equivalence class of r and set $\mathsf{D}^A_\Gamma = \hat{\mathsf{D}}^A_\Gamma = \mathsf{Tm}/{=_{\beta\eta}}$. Again, D, with application and evaluation defined as in the last section, constitutes a syntactical applicative System F structure.

Long normal forms are characterized by the two mutually defined judgments

$$\begin{array}{ll} \Gamma \vdash t \Uparrow A & t \text{ is a long normal form of type } A \\ \Gamma \vdash t \Downarrow A & t \text{ is a neutral long normal form of type } A \end{array}$$

given by the following rules:

$$\frac{}{\Gamma \vdash x \Downarrow \Gamma(x)} \qquad \frac{\Gamma \vdash r \Downarrow A \to B \qquad \Gamma \vdash s \Uparrow A}{\Gamma \vdash r\, s \Downarrow B} \qquad \frac{\Gamma \vdash r \Downarrow \forall X A}{\Gamma \vdash rB \Downarrow A[B/X]}$$

$$\frac{\Gamma \vdash r \Downarrow X}{\Gamma \vdash r \Uparrow X} \qquad \frac{\Gamma, x{:}A \vdash t \Uparrow B}{\Gamma \vdash \lambda x{:}A.\, t \Uparrow A \to B} \qquad \frac{\Gamma \vdash t \Uparrow A}{\Gamma \vdash \Lambda X t \Uparrow \forall X A}\ X \notin \mathsf{FV}(\Gamma)$$

From the model construction of System F we want to harvest that each well-typed term has a η-long β-normal form. Thus, we define an interpretation space by setting

$$\begin{array}{l} \Gamma \vdash d \sim d' \in \overline{A} \iff \text{exists } r \text{ with } d = d' = \overline{r} \text{ and } \Gamma \vdash r \Uparrow A, \\ \Gamma \vdash d \sim d' \in \underline{A} \iff \text{exists } r \text{ with } d = d' = \overline{r} \text{ and } \Gamma \vdash r \Downarrow A. \end{array}$$

Lemma 7 (Weakening). $\underline{A}, \overline{A} \in \mathbb{K}^A$.

Indeed, $\underline{A}$ and $\overline{A}$ span an interpretation space, which we will prove in detail in the following.

Lemma 8 (Interpretation space)

K-FUN-E	$\underline{A \to B}$	$\subseteq \overline{A} \to \underline{B}$	
K-FUN-I	$\underline{A} \to \overline{B}$	$\subseteq \overline{A \to B}$	
K-ALL-E	$\underline{\forall X A}$	$\subseteq B.\underline{A[B/X]}$	
K-ALL-I	$X.\overline{A[X/Y]}$	$\subseteq \overline{\forall Y A}$	*for a new* X

Proof. K-FUN-E $\underline{A \to B} \subseteq \overline{A} \to \underline{B}$

$\Gamma \vdash f \sim f' \in \underline{A \to B}$	by hyp.
$f = f' = \overline{r}$ and $\Gamma \vdash r \Downarrow A \to B$	by def.
$\Gamma' \leq \Gamma$ and $\Gamma' \vdash d \sim d' \in \overline{A}$	assumption
$d = d' = \overline{s}$ and $\Gamma' \vdash s \Uparrow A$	by def.
$f \cdot d = f' \cdot d' = \overline{r\,s}$	def. of application
$\Gamma' \vdash r\,s \Downarrow B$	rule, Lemma 7
$\Gamma' \vdash f \cdot d \sim f' \cdot d' \in \underline{B}$	by def.
$\Gamma \vdash f \sim f' \in \overline{A} \to \underline{B}$	since Γ', d, d' arb.

K-FUN-I $\underline{A} \to \overline{B} \subseteq \overline{A \to B}$

$\Gamma \vdash \overline{r} \sim \overline{r'} \in \underline{A} \to \overline{B}$	by hyp.
$\Gamma, x\!:\!A \vdash x \Downarrow A$	rule
$\Gamma, x\!:\!A \vdash \overline{x} \sim \overline{x} \in \underline{A}$	by def.
$\Gamma, x\!:\!A \vdash \overline{r} \cdot \overline{x} \sim \overline{r'} \cdot \overline{x} \in \overline{B}$	by def. $\to$
$\overline{r\,x} = \overline{r'\,x} = \overline{t}$ and $\Gamma, x\!:\!A \vdash t \Uparrow B$	by def.
$\overline{r} = \overline{\lambda x\!:\!A.\, r\,x} = \overline{r'} = \overline{\lambda x\!:\!A.\, r'\,x} = \overline{\lambda x\!:\!A.\, t}$	η
$\Gamma \vdash \lambda x\!:\!A.\, t \Uparrow A \to B$	rule
$\Gamma \vdash \overline{r} \sim \overline{r'} \in \overline{A \to B}$	by def.

K-ALL-E $\underline{\forall X A} \subseteq B.\underline{A[B/X]}$

$\Gamma \vdash d \sim d' \in \underline{\forall X A}$	assumption
$d = d' = \overline{r}$ and $\Gamma \vdash r \Downarrow \forall X A$	by def.
$d \cdot B = d' \cdot B = \overline{r\,B}$ and $\Gamma \vdash r\,B \Downarrow A[B/X]$	rule, app.
$\Gamma \vdash d \cdot B \sim d' \cdot B \in \underline{A[B/X]}$	by def.
$\Gamma \vdash d \sim d' \in B.\underline{A[B/X]}$	by def. $B.\mathcal{A}$

K-ALL-I $X.\overline{A[X/Y]} \subseteq \overline{\forall Y\, A}$ for a new X.

$$
\begin{array}{lr}
\Gamma \vdash \overline{r} \sim \overline{r'} \in X.\overline{A[X/Y]} & \text{assumption} \\
\Gamma \vdash \overline{r} \cdot X \sim \overline{r'} \cdot X \in \overline{A[X/Y]} & \text{by def. } X.\mathcal{A} \\
\overline{r\,X} = \overline{r'\,X} = \overline{t} \text{ and } \Gamma \vdash t \Uparrow A[X/Y] & \text{by def.} \\
\overline{r} = \overline{\Lambda X.\, r\,X} = \overline{r'} = \overline{\Lambda X.\, r'\,X} = \overline{\Lambda X t} & \eta \\
X \notin \mathsf{FV}(\Gamma) & \text{since } X \text{ new} \\
\Gamma \vdash \Lambda X t \Uparrow \forall X.\, A[X/Y] = \forall Y A & \text{rule} \\
\Gamma \vdash \overline{r} \sim \overline{r'} \in \overline{\forall Y A} & \text{by def.}
\end{array}
$$

The rest is just an application of the fundamental theorem. Recall that η_{id} is the identity map, and let this time $\rho_{\mathsf{id}}(X) = \underline{X}$ for all $X \in \mathsf{TyVar}$. Again, $\eta_{\mathsf{id}} \Vdash \rho_{\mathsf{id}}$, and $\Gamma \vdash \overline{\eta_{\mathsf{id}}} \sim \overline{\eta_{\mathsf{id}}} \in [\![\Gamma]\!]_{\rho_{\mathsf{id}}}$.

Theorem 4 (Weak $\beta\eta$-normalization of System F). *If $\Gamma \vdash t : A$ then t β-reduces η-expands to a long normal form t'.*

Proof. Clearly, $A \Vdash [\![A]\!]_{\rho_{\mathsf{id}}}$. By Thm. 2, $\Gamma \vdash (\!|t|\!)_{\overline{\eta_{\mathsf{id}}}} \sim (\!|t|\!)_{\overline{\eta_{\mathsf{id}}}} \in [\![A]\!]_{\rho_{\mathsf{id}}}$, meaning $t =_{\beta\eta} t'$ with $\Gamma \vdash t' \Uparrow A$. We conclude by Church-Rosser for β-reduction η-expansion [17].

7 Normalization by Evaluation

In this section we now define a normalization function which maps exactly the $\beta\eta$-equal terms of the same type to the same η-long β-normal form. To this end, we define judgmental $\beta\eta$-equality $\Gamma \vdash t = t' : A$ and the function $\mathsf{nf}(\Gamma \vdash t : A)$ such that it is

1. complete for judgmental equality, i. e., $\Gamma \vdash t = t' : A$ implies $\mathsf{nf}(\Gamma \vdash t : A) \equiv \mathsf{nf}(\Gamma \vdash t' : A)$, and
2. sound, i. e., if $\Gamma \vdash t : A$ then $\Gamma \vdash t = \mathsf{nf}(\Gamma \vdash t : A) : A$.

7.1 Judgmental Equality

Judgmental $\beta\eta$-equality $\Gamma \vdash t = t' : A$ is defined inductively by the following axiom, plus congruence rules and equivalence rules (reflexivity, symmetry, and transitivity).

$$
\frac{\Gamma, x : A \vdash t : B \qquad \Gamma \vdash s : A}{\Gamma \vdash (\lambda x : A.\, t)\, s = t[s/x] : B}
\qquad
\frac{\Gamma \vdash t : A \to B}{\Gamma \vdash \lambda x : A.\, t\, x = t : A \to B}\; x \notin \mathsf{FV}(t)
$$

$$
\frac{\Gamma \vdash t : A \qquad X \notin \mathsf{FV}(\Gamma)}{\Gamma \vdash (\Lambda X t)\, B = t[B/X] : A[B/X]}
\qquad
\frac{\Gamma \vdash t : \forall X A}{\Gamma \vdash \Lambda X.\, t\, X = t : \forall X A}\; X \notin \mathsf{FV}(t)
$$

A fundamental theorem for judgmental equality is of course not valid for arbitrary *Kripke relations*, it can only be shown for *Kripke PERs*, since symmetry and transitivity have to be modeled. Also, to model the above equations, the evaluation function must satisfy additional laws:

DEN-SUBST	$(\!\|t[s/x]\|\!)_\eta = (\!\|t\|\!)_{\eta[x\mapsto(\!\|s\|\!)_\eta]}$	if $\Gamma, x\!:\!A \vdash t : B$ and $\Gamma \vdash s : A$
DEN-TY-SUBST	$(\!\|t[A/X]\|\!)_\eta = (\!\|t\|\!)_{\eta[X\mapsto A\eta]}$	if $\Gamma \vdash t : B$ and $X \notin \mathsf{FV}(\Gamma)$
DEN-IRR	$(\!\|t\|\!)_\eta = (\!\|t\|\!)_{\eta'}$	if $\eta(x) = \eta'(x)$ for all $x \in \mathsf{FV}(t)$

If these laws are satisfied, $(\mathsf{D}, \_\cdot\_, (\!|\_|\!)\_)$ is called a *syntactical combinatorial System F algebra* (cf. [1]).

For the following theorem, consider an interpretation space of Kripke-PERs over a syntactical combinatorial System F algebra.

Theorem 5 (Validity of equality). *Let $\sigma \Vdash \rho$ and $\Delta \vdash \eta \sim \eta' \in [\![\Gamma]\!]_\rho$. If $\Gamma \vdash t = t' : A$ then $\Delta \vdash (\!\|t\|\!)_\eta \sim (\!\|t'\|\!)_{\eta'} \in [\![A]\!]_\rho$.*

Proof. By induction on $\Gamma \vdash t = t' : A$.

7.2 The Normalization Algorithm

Normalization by evaluation consists of two steps: first, evaluate the term in the identity environment, obtaining a semantic object, and then, reify the semantic object back to syntax, yielding a long normal form.

Evaluation. For the evaluation we need a combinatorial algebra Val with computable application and evaluation and with variables. One possibility is to let Val be the solution of the recursive domain equation:

$$\mathsf{Val} = (\mathsf{Var} \times (\mathsf{Val} \cup \mathsf{Ty})^{<\omega} + [\mathsf{Val} \to \mathsf{Val}] + (\mathsf{Ty} \to \mathsf{Val}))_\perp.$$

Then a semantic object $d \in \mathsf{Val}$ is either a neutral object e of the shape $e ::= x \mid e\,d \mid e\,A$, a continuous function $f \in [\mathsf{Val} \to \mathsf{Val}]$ on semantic objects, a function $F \in \mathsf{Ty} \to \mathsf{Val}$ from types to semantic objects, or undefined, $\perp$.

Let $\mathsf{D}^A_\Gamma = \mathsf{Val}$ for all Γ, A. Application is defined by

$$\begin{array}{ll} e \cdot d = e\,d & e \cdot A = e\,A \\ f \cdot d = f(d) & F \cdot A = F(A), \end{array}$$

yielding $\perp$ in all other cases, and evaluation by DEN-VAR, DEN-FUN-E, DEN-ALL-E and

$$\begin{array}{ll} (\!|\lambda x\!:\!A.\, t|\!)_\eta(d) & = (\!|t|\!)_{\eta[x\mapsto d]} \\ (\!|\Lambda X t|\!)_\eta(A) & = (\!|t|\!)_{\eta[X\mapsto A]}. \end{array}$$

It is easy to check that $(\mathsf{D}, \_\cdot\_, (\!|\_|\!)\_)$ forms a syntactical combinatorial System F algebra.

Reification converts semantic objects back to expressions. Functions are reified by applying them to fresh variables. This is of course only possible if Val contains the variables.

We adapt *contextual reification* [2] to System F and define inductively the mutual judgements

$$\begin{array}{ll} \Gamma \vdash d \searrow t \Uparrow A & d \text{ reifies to } t \text{ at type } A, \\ \Gamma \vdash d \searrow t \Downarrow A & d \text{ reifies to } t, \text{ inferring type } A. \end{array}$$

These judgements enrich the corresponding judgements for long normal forms by the semantic object d to be reified to term t. As a consequence, the output t is trivially in long normal form.

$$\frac{}{\Gamma \vdash x \searrow x \Downarrow \Gamma(x)}$$

$$\frac{\Gamma \vdash e \searrow r \Downarrow A \to B \qquad \Gamma \vdash d \searrow s \Uparrow A}{\Gamma \vdash e\, d \searrow r\, s \Downarrow B} \qquad \frac{\Gamma \vdash e \searrow r \Downarrow \forall X A}{\Gamma \vdash e\, B \searrow r\, B \Downarrow A[B/X]}$$

$$\frac{\Gamma \vdash e \searrow r \Downarrow X}{\Gamma \vdash e \searrow r \Uparrow X}$$

$$\frac{\Gamma, x{:}A \vdash f \cdot x \searrow t \Uparrow B}{\Gamma \vdash f \searrow \lambda x{:}A.\, t \Uparrow A \to B} \qquad \frac{\Gamma \vdash F \cdot X \searrow t \Uparrow A}{\Gamma \vdash F \searrow \Lambda X t \Uparrow \forall X A}\ X \notin \mathsf{FV}(\Gamma).$$

These rules can be interpreted computationally by considering them clauses of a logic program. Both judgements take Γ and d as input and return t. In the type-directed mode $\Uparrow$, type A is input, and in the inference mode $\Downarrow$, type A is output. It is easy to check that the associated logic program is well-moded [13]. Termination, however, will follow from the fundamental theorem.

Lemma 9 (Weakening). *Let $\Gamma' \leq \Gamma$.*

1. *If $\Gamma \vdash e \searrow r \Downarrow A$ then $\Gamma' \vdash e \searrow r \Downarrow A$.*
2. *If $\Gamma \vdash d \searrow t \Uparrow A$ then $\Gamma' \vdash d \searrow t \Uparrow A$.*

7.3 Completeness of NbE

We obtain completeness (and termination) of the normalization function as instance of the fundamental theorem for judgmental equality. Let an interpretation space be defined by

$$\begin{array}{ll} \Gamma \vdash d \sim d' \in \overline{A} \iff & \text{exists } t \text{ with } \Gamma \vdash d \searrow t \Uparrow A \text{ and } \Gamma \vdash d' \searrow t \Uparrow A, \\ \Gamma \vdash d \sim d' \in \underline{A} \iff & \text{exists } t \text{ with } \Gamma \vdash d \searrow t \Downarrow A \text{ and } \Gamma \vdash d' \searrow t \Downarrow A. \end{array}$$

Lemma 10 (Interpretation space). *$\underline{A}, \overline{A}$ form an interpretation space of Kripke PERs.*

Proof. Analogous to Lemma 8.

Theorem 6 (Completeness of NbE). *If* $\Gamma \vdash t = t' : A$ *then* $\Gamma \vdash (\!|t|\!)_{\eta_{\mathsf{id}}} \searrow r \Uparrow A$ *and* $\Gamma \vdash (\!|t'|\!)_{\eta_{\mathsf{id}}} \searrow r \Uparrow A$ *for some long normal form* r.

Proof. Since $\Gamma \vdash \eta_{\mathsf{id}} \sim \eta_{\mathsf{id}} \in [\![\Gamma]\!]_{\rho_{\mathsf{id}}}$, by Thm. 5 $\Gamma \vdash (\!|t|\!)_{\eta_{\mathsf{id}}} \sim (\!|t'|\!)_{\eta_{\mathsf{id}}} \in \overline{A}$.

The normalization function $\mathsf{nf}(\Gamma \vdash t : A)$ can now be defined to yield the t' such that $\Gamma \vdash (\!|t|\!)_{\eta_{\mathsf{id}}} \searrow t' \Uparrow A$.

7.4 Soundness of NbE

Soundness (and termination) of the normalization function is a consequence of the fundamental theorem for typing, applied to a Kripke relation between semantics and syntax. With D defined as above, we set $\hat{\mathsf{D}}^A_\Gamma = \mathsf{Tm}^A_\Gamma / (\Gamma \vdash \_ = \_ : A)$, i.e., terms modulo judgmental equality, which forms a syntactical applicative System F structure by virtue of

$$\begin{aligned} \mathsf{app}^{A,B}(\overline{r}, \overline{s}) &= \overline{r\, s} \\ \mathsf{App}^{X.A}(\overline{r}, B) &= \overline{r\, B} \\ (\!|t|\!)_{\overline{\sigma}} &= \overline{t\sigma} \end{aligned}$$

Note that the *typed* applicative structure is crucial here, on untyped terms modulo judgmental equality one cannot define a total application operation. We let

$$\begin{aligned} \Gamma \vdash d \sim \overline{t} \in \overline{A} &\iff \text{exists } t' \text{ with } \Gamma \vdash d \searrow t' \Uparrow A \text{ and } \Gamma \vdash t = t' : A, \\ \Gamma \vdash d \sim \overline{t} \in \underline{A} &\iff \text{exists } t' \text{ with } \Gamma \vdash d \searrow t' \Downarrow A \text{ and } \Gamma \vdash t = t' : A. \end{aligned}$$

Lemma 11 (Interpretation space). $\underline{A}, \overline{A}$ *form an interpretation space.*

Theorem 7 (Soundness of NbE). *If* $\Gamma \vdash t : A$ *then* $\Gamma \vdash (\!|t|\!)_{\eta_{\mathsf{id}}} \searrow t' \Uparrow A$ *and* $\Gamma \vdash t = t' : A$.

Proof. For all $(x : B) \in \Gamma$, it holds that $\Gamma \vdash x \sim \overline{x} \in \underline{B}$, hence, $\Gamma \vdash \eta_{\mathsf{id}}(x) \sim \overline{\eta_{\mathsf{id}}}(x) \in [\![B]\!]_{\rho_{\mathsf{id}}}$, thus, $\Gamma \vdash \eta_{\mathsf{id}} \sim \overline{\eta_{\mathsf{id}}} \in [\![\Gamma]\!]_{\rho_{\mathsf{id}}}$. By Thm. 2, $\Gamma \vdash (\!|t|\!)_{\eta_{\mathsf{id}}} \sim \overline{t} \in [\![A]\!]_{\rho_{\mathsf{id}}}$. We conclude by $[\![A]\!]_{\rho_{\mathsf{id}}} \subseteq \overline{A}$.

Summarizing this section, we have obtained a $\beta\eta$-normalization function for System F which is complete and sound for judgmental equality. We have crucially used that the fundamental theorem for typing is not restricted to PER semantics but has been formulated for Kripke relations between two different applicative System F structures.

8 Conclusion

We have introduced the concept of type *interpretation space* and proven generic fundamental theorems for typing and judgmental equality in System F. As instances, we obtained proofs of weak normalization for β and $\beta\eta$, and proofs of soundness and completeness for a normalization-by-evaluation algorithm based on contextual reification.

Further work. We seek to extend this work to type theories with non-trivial equality on the type level, like System F^{ω} and the Calculus of Constructions.

Acknowledgments. This work was carried out during a visit to Frédéric Blanqui and Cody Roux at LORIA, Nancy, France, financed by the *Bayerisch-Französisches Hochschulzentrum*. My gratitude extends also to Thierry Coquand and Peter Dybjer for discussions on the topic, and to Ralph Matthes, Cody Roux, and the anonymous referees for their suggestions which helped to improve this paper.

References

1. Abel, A., Coquand, T., Dybjer, P.: Normalization by evaluation for Martin-Löf Type Theory with typed equality judgements. In: Proc. of the 22nd IEEE Symp. on Logic in Computer Science (LICS 2007), pp. 3–12. IEEE Computer Soc. Press, Los Alamitos (2007)
2. Abel, A., Coquand, T., Dybjer, P.: Verifying a semantic $\beta\eta$-conversion test for martin-löf type theory. In: Audebaud, P., Paulin-Mohring, C. (eds.) MPC 2008. LNCS, vol. 5133, pp. 29–56. Springer, Heidelberg (2008)
3. Altenkirch, T., Dybjer, P., Hofmann, M., Scott, P.J.: Normalization by evaluation for typed lambda calculus with coproducts. In: Proc. of the 16th IEEE Symp. on Logic in Computer Science (LICS 2001), pp. 303–310. IEEE Computer Soc. Press, Los Alamitos (2001)
4. Altenkirch, T., Hofmann, M., Streicher, T.: Reduction-free normalisation for a polymorphic system. In: Proc. of the 11th IEEE Symp. on Logic in Computer Science (LICS 1996), pp. 98–106. IEEE Computer Soc. Press, Los Alamitos (1996)
5. Altenkirch, T., Hofmann, M., Streicher, T.: Reduction-free normalisation for System F (1997), http://www.cs.nott.ac.uk/~{}txa/publ/f97.pdf
6. Barendregt, H.: The Lambda Calculus: Its Syntax and Semantics. North Holland, Amsterdam (1984)
7. Berger, U., Schwichtenberg, H.: An inverse to the evaluation functional for typed λ-calculus. In: Proc. of the 6th IEEE Symp. on Logic in Computer Science (LICS 1991), pp. 203–211. IEEE Computer Soc. Press, Los Alamitos (1991)
8. Blanqui, F.: Definitions by rewriting in the calculus of constructions. Mathematical Structures in Computer Science 15, 37–92 (2005)
9. Blanqui, F., Jouannaud, J.-P., Strub, P.-Y.: Building decision procedures in the calculus of inductive constructions. In: Duparc, J., Henzinger, T.A. (eds.) CSL 2007. LNCS, vol. 4646, pp. 328–342. Springer, Heidelberg (2007)
10. Chrzaszcz, J., Walukiewicz-Chrzaszcz, D.: Towards rewriting in coq. In: Comon-Lundh, H., Kirchner, C., Kirchner, H. (eds.) Jouannaud Festschrift. LNCS, vol. 4600, pp. 113–131. Springer, Heidelberg (2007)
11. Constable, R.: Team: Implementing Mathematics with the Nuprl Proof Development System. Prentice Hall, Englewood Cliffs (1986)
12. Danvy, O.: Type-directed partial evaluation. In: Hatcliff, J., Mogensen, T.Æ., Thiemann, P. (eds.) DIKU 1998. LNCS, vol. 1706, pp. 367–411. Springer, Heidelberg (1999)
13. Debray, S.K., Warren, D.S.: Automatic mode inference for logic programs. Journal of Logic Programming 5, 207–229 (1988)

14. Filinski, A.: A semantic account of type-directed partial evaluation. In: Nadathur, G. (ed.) PPDP 1999. LNCS, vol. 1702, pp. 378–395. Springer, Heidelberg (1999)
15. Girard, J.-Y., Lafont, Y., Taylor, P.: Proofs and Types. Cambridge Tracts in Theoretical Computer Science, vol. 7. Cambridge University Press, Cambridge (1989)
16. INRIA: The Coq Proof Assistant, Version 8.1. INRIA (2007), http://coq.inria.fr/
17. Jay, B.: Long $\beta\eta$ normal forms and confluence. Technical Report ECS-LFCS-91-183, University of Edinburgh (1991)
18. Matthes, R.: Extensions of System F by Iteration and Primitive Recursion on Monotone Inductive Types. Ph.D. thesis, Ludwig-Maximilians-University (1998)
19. Norell, U.: Towards a practical programming language based on dependent type theory. Ph.D. thesis, Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden (2007)
20. Pierce, B.C.: Types and Programming Languages. MIT Press, Cambridge (2002)
21. Pollack, R.: The Theory of LEGO. Ph.D. thesis, University of Edinburgh (1994)
22. Streicher, T.: Semantics of Type Theory. Progress in Theoretical Computer Science. Birkhaeuser Verlag, Basel (1991)
23. Vaux, L.: A type system with implicit types, English version of his mémoire de maîtrise (2004)

Variable Dependencies of Quantified CSPs*

Marko Samer

Department of Computer Science
TU Darmstadt, Germany
samer@cs.tu-darmstadt.de

Abstract. Quantified constraint satisfaction extends classical constraint satisfaction by a linear order of the variables and an associated existential or universal quantifier to each variable. In general, the semantics of the quantifiers does not allow to change the linear order of the variables arbitrarily without affecting the truth value of the instance. In this paper we investigate variable dependencies that are caused by the influence of the relative order between these variables on the truth value of the instance. Several approaches have been proposed in the literature for identifying such dependencies in the context of quantified Boolean formulas. We generalize these ideas to quantified constraint satisfaction and present new concepts that allow a refined analysis.

1 Introduction

Constraint satisfaction provides a formal framework for a large class of combinatorial problems. For a set of constraints over a set of variables with respective domains, the constraint satisfaction problem (CSP) is to decide whether there exists an instantiation of the variables such that all constraints are simultaneously satisfied. In the case of quantified constraint satisfaction, the set of constraints is accompanied by a linear order of the variables and an existential or universal quantifier associated with each variable. The quantified constraint satisfaction problem (QCSP) is then to decide whether the constraints are simultaneously satisfied by instantiations according to the linear order of the variables and the associated quantifiers (see Section 2 for a formal definition). If the answer is affirmative, we say the instance is true; otherwise, we say it is false. Classical constraint satisfaction can be seen as a special case of quantified constraint satisfaction where all variables are existentially quantified. Evidently, in this case the linear order of the variables (that is, the order in which the variables are instantiated) can be arbitrarily changed without affecting the truth value of the instance. However, if we also allow universal quantifiers, changing the relative order of two variables may affect the truth value of the instance.

In this paper we investigate variable dependencies of QCSPs, i.e., dependencies between variables that arise from the relative order between these variables and their associated quantifiers. Intuitively, we say two variables are independent from each other if their relative order has no influence on the truth value of the instance; otherwise,

* This research was carried out during the author's postdoc position at the University of Durham and was supported by the EPSRC, project EP/E001394/1.

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 512–527, 2008.

we say they are dependent. For example, two equally quantified variables that are adjacent in the linear order are trivially independent from each other. We write $\mathcal{Q}\mathcal{C}$ to denote a QCSP instance, where $\mathcal{C}$ is a set of constraints and $\mathcal{Q}$ is the *quantifier prefix* of the form $\mathsf{Q}_1 x_1\, \mathsf{Q}_2 x_2 \ldots \mathsf{Q}_n x_n$ representing the linear order of the variables and their respective quantifier $\mathsf{Q}_i \in \{\forall, \exists\}$. It is well known that the number of quantifier alternations in the prefix is an indicator for the problem hardness. In particular, the special case of quantified Boolean formulas (QBFs), i.e., the subclass of QCSPs where all variables have Boolean domain and the constraints are clauses, yields complete problems for every class in the polynomial hierarchy by bounding the number of quantifier alternations respectively; in the case of unbounded quantifier alternations, it yields a PSPACE-complete problem. It is thus not surprising that, already for QBFs, identifying variable dependencies is in general PSPACE-hard [15]. Therefore, we focus on independencies that can be recognized in polynomial time, i.e., we assume by default that two variables depend on each other if we cannot tell whether they are independent.

Most research on QCSPs over the last years concentrated on theoretical issues, primarily on the complexity of QCSP subclasses (see, e.g., [5,7,9,13]). But there also has been made remarkable progress in the development of efficient QBF solvers [3,4,14], and recently even solvers for QCSPs that work without an encoding into QBFs have gained particular interest [6,12].

In this paper we follow a number of works on identifying variable dependencies [4,8,14,15] and consider this question from a more theoretical point of view. To the best of our knowledge, Biere [4] was the first who identified variable dependencies in our sense. In the expansion step of his QBF solver, Biere applies Shannon expansion in order to eliminate universal variables. This, however, requires that clauses containing existential variables succeeding the expanded universal variable in the quantifier prefix have to be duplicated. Actually, it suffices to duplicate only those clauses that contain existential variables that depend on the universal variable. Thus, since it is desirable to keep the size of the formula small, Biere proposed an efficient method for identifying independent variables that do not need to be taken into account for clause duplication. Closely related to Biere's approach is the work of Ayari and Basin [1] and quantifier shifting rules as investigated by Egly, Tompits, and Woltran [11].

Bubeck and Kleine Büning [8] demonstrated how to overcome the restriction to innermost universal variables in Biere's approach. At the same time, Samer and Szeider [15] presented another generalization and proposed the generic framework of *dependency schemes* for identifying variable dependencies. Their research, however, was motivated by backdoor set detection for QBFs. Backdoor sets allow to identify tractable classes of QBFs with an unbounded number of quantifier alternations. Crucial in this context is the bounded size of the backdoor sets, which in turn is closely related to the question of variable dependencies. Recently, Lonsing and Biere [14] considered a further application: Variable dependencies for QBFs in negation normal form.

We follow this line of research and extend it in three main areas:

1. We propose a formal definition of variable independence and strengthen the notion of dependency schemes by using this new definition. Previous research is based on a rather intuitive understanding of independence, but, to the best of our knowledge, no formal definition has been presented so far.

2. We generalize previous methods for identifying variable dependencies from QBFs to QCSPs. In the case of the standard dependency scheme (Section 3.1), this can be done in a straightforward way, while in the case of the triangle dependency scheme (Section 3.2), we have to introduce more general notions.
3. We generalize the idea of the triangle dependency scheme to an infinite hierarchy of triangle dependency schemes. We show that each dependency scheme in this hierarchy is strictly stronger than the one on the previous level. Moreover, we present the *generalized triangle dependency scheme* as the closure of this hierarchy.

Note that since our results hold for QCSPs in general, they also hold for important subclasses like QBFs. As mentioned above, the question about variable dependencies has already been shown to be relevant in the context of expansion-based QBF solvers and for the identification of tractable classes of QBFs via backdoor sets. We believe that the knowledge about variable dependencies may also be exploited in other related areas.

This paper is organized as follows: In Section 2, we introduce the basic notions. In Section 3, we present our refined framework of dependency schemes. Then, in Sections 3.1 and 3.2, we consider the standard dependency scheme and the triangle dependency hierarchy based on our new framework. Finally, we conclude in Section 4.

2 Preliminaries

A *constraint network* $\mathcal{C}$ is a set of constraints over a set of variables V with respective domains $dom(x)$, $x \in V$. A *constraint* $C \in \mathcal{C}$ defines which instantiations of variables in its *scope* $var(C) \subseteq V$ by their respective domain elements satisfy C (note that we do not require an extensional representation of the constraint relations). We refer to $|var(C)|$ as the *arity* of C and we put $var(\mathcal{C}) = \bigcup_{C \in \mathcal{C}} var(C)$. A *quantified constraint satisfaction formula* (*QCSP formula*, for short) φ is of the form $\mathcal{Q}\mathcal{C}$, where $\mathcal{C}$ is a constraint network and $\mathcal{Q}$ is the *quantifier prefix* of the form $\mathsf{Q}_1 x_1 \ldots \mathsf{Q}_n x_n$ with $\mathsf{Q}_i \in \{\forall, \exists\}$ and $x_i \in var(\mathcal{C})$ for all $1 \leq i \leq n$. We put $var(\varphi) = var(\mathcal{C})$ and we assume w.l.o.g. that each variable in $var(\mathcal{C})$ occurs exactly once in the quantifier prefix. We call a constraint *binary* if it has arity two and we call a variable *Boolean* if its domain contains exactly two elements.

For a QCSP formula $\varphi = \mathsf{Q}_1 x_1 \ldots \mathsf{Q}_n x_n \, \mathcal{C}$ and a variable $x_i \in var(\varphi)$, we define the *depth* of x_i in φ by $\delta_\varphi(x_i) = i$ and we put $q_\varphi(x_i) = \mathsf{Q}_i$. Moreover, we define $var_\mathsf{Q}(\varphi) = \{x \in var(\varphi) : q_\varphi(x) = \mathsf{Q}\}$. We also write $R_\varphi(x) = \{z \in var(\varphi) : \delta_\varphi(x) < \delta_\varphi(z)\}$ for the set of variables on the right of x in the quantifier prefix and $R^\diamond_\varphi(x) = \{z \in R_\varphi(x) : \exists v \in R_\varphi(x),\, q_\varphi(v) \neq q_\varphi(x),\, \delta_\varphi(v) \leq \delta_\varphi(z)\}$ for the set of variables on the right of x starting at the first variable (from left to right) with different quantification; $L_\varphi(x)$ and $L^\diamond_\varphi(x)$ are defined symmetrically.

An *assignment* on some set $X \subseteq var(\varphi)$ of variables is a mapping $\tau : X \to \bigcup_{x \in X} dom(x)$ such that $\tau(x) \in dom(x)$ for all $x \in X$; we put $dom(\tau) = X$. An assignment τ on X *satisfies* (*falsifies*) a constraint C if every instantiation of the variables in $var(C)$ that coincides with τ on $X \cap var(C)$ satisfies (does not satisfy) C. For an assignment τ on X and a constraint C, we denote by $C[\tau]$ the constraint with scope $var(C[\tau]) = var(C) \setminus X$ such that an assignment σ on $var(C) \setminus X$ satisfies

$C[\tau]$ if and only if $\tau \cup \sigma$ satisfies C. For a constraint network $\mathcal{C}$, we put $\mathcal{C}[\tau] = \{C[\tau] : C \in \mathcal{C},\ \tau \text{ does not satisfy } C\}$. For a QCSP formula φ with constraint network $\mathcal{C}$, we write $\varphi[\tau]$ to denote the QCSP formula obtained from φ by replacing $\mathcal{C}$ with $\mathcal{C}[\tau]$ and removing all superfluous quantifications.

The following definition is motivated by Benedetti [2]: An *assignment tree* $\mathcal{T} = (T, \lambda)$ on some set $X \subseteq var(\varphi)$ of variables is a pair of a rooted tree T where all leaves have depth $|X| + 1$ and a function λ labeling every node t (except the root) of T with a pair (x, d) such that $x \in X$ and $d \in dom(x)$; if $x \in var_\exists(\varphi)$, then t has no siblings (i.e., t's parent has only one child), and if $x \in var_\forall(\varphi)$, then t has $|dom(x)| - 1$ siblings and there is no sibling t' with $\lambda(t') = \lambda(t)$ (i.e., t's parent has exactly one child for each $d \in dom(x)$). Moreover, if t_1 and t_2 are two nodes of T such that t_1 has lower depth than t_2 and it holds that $\lambda(t_1) = (x, d)$ and $\lambda(t_2) = (y, d')$, then $\delta_\varphi(x) < \delta_\varphi(y)$. Every leaf t of T corresponds to an assignment τ on X such that for every node t' (except the root) on the path from the root to t it holds that $\lambda(t') = (x, \tau(x))$, where $x \in X$. We simply write $\tau \in \mathcal{T}$ for such an assignment. For $x \in X$ and $d \in dom(x)$, we write $\mathcal{T}_{x=d}$ to denote the assignment subtree of $\mathcal{T}$ with root t such that t is a child of the root of $\mathcal{T}$ and $\lambda(t) = (x, d)$.

Note that every linear ordering on the domain elements of each variable gives rise to a linear ordering of the nodes $t_1, t_2, \ldots, t_n$ with the same depth in an assignment tree. In this case we write $\mathcal{T}[x]$ to denote the corresponding unique sequence $d_1, d_2, \ldots, d_n$ of domain elements such that $\lambda(t_i) = (x, d_i)$.

A QCSP formula φ with constraint network $\mathcal{C}$ is *true* (or *satisfiable*) if there exists an assignment tree $\mathcal{T}$ on $var(\varphi)$ such that $\mathcal{C}[\tau] = \emptyset$ for all $\tau \in \mathcal{T}$ (in this case we call $\mathcal{T}$ a *satisfying assignment tree* or a *model* of φ); otherwise φ is *false* (or *unsatisfiable*). The *quantified constraint satisfaction problem QCSP* is to decide whether φ is true. Two QCSP formulas φ and ψ are (satisfiability) *equivalent* if they are either both true or both false.

3 Dependency Schemes

Let us first clarify what we exactly mean when we say that two variables depend on each other or are independent from each other. The motivation for our definition arises from the intuitive meaning of independence in the related literature: Two variables are considered independent if their relative order in the quantifier prefix does not affect the truth value of the formula. This intuition is easy to formalize if we consider only adjacent variables x and y in the quantifier prefix: Just swap x and y and check whether the truth value has changed. However, if x and y are not adjacent, things become more complicated since simply swapping x and y also changes the relative order to the variables between x and y. Then, however, it is not clear at all which change of the relative order causes the change of the truth value or whether two changes of the relative order cancel each other out with respect to the truth value of the formula.

Since we are interested in a notion of independence for any two variables x and y (not only adjacent ones), we have to develop more general criteria. To this aim, assume that x is universal and y is existential; if they are equally quantified, we consider them always independent from each other. If the formula is false and y is on the right of x in

the quantifier prefix, i.e., the values assigned to y can be different for different values assigned to x, it is easy to see that the formula cannot become true if we shift y to the left of x where it can take only one value for each value assigned to x. Symmetrically, if the formula is true and y is on the left of x in the quantifier prefix, i.e., y can take only one value for each value assigned to x, it is easy to see that the formula cannot become false if we shift y to the right of x where the values assigned to y can be different for different values assigned to x. Thus, there are only two cases left that allow us to distinguish between dependence and independence:

(i) If y is on the right of x in the quantifier prefix and the formula is true, we consider y independent from x if the formula remains true under the restriction that the values assigned to y must be the same for all values assigned to x. In other words, we require that if there exists any satisfying assignment tree, then there exists a satisfying assignment tree such that the assignments to y are the same in all subtrees for different assignments to x. Figure 1 illustrates this condition with an arbitrary satisfying assignment tree (left) and a possible restricted satisfying assignment tree (right), where the values assigned to y are the same in both subtrees for two different values assigned to x. According to the definitions in Section 2 and in order to avoid confusion when referring to nodes of an assignment tree, note that the dotted lines indicate nodes labeled with values assigned to the corresponding variables.

(ii) If y is on the left of x in the quantifier prefix and the formula is false, we consider y independent from x if the formula remains false under the relaxation that the values assigned to y can be different for different values assigned to x. The problem in this case, however, is to express this kind of relaxation in terms of assignment trees since y is on the left of x and thus cannot take different values for different values assigned to x. Therefore, we transform the original formula φ into a formula φ_x by shifting x to the left most position in the quantifier prefix. Then there exists a satisfying assignment tree of φ if and only if there exists a satisfying assignment tree of φ_x under the restriction that the values assigned to variables that have been on the left of x in φ are the same in all subtrees for different assignments to x. In this way we can easily express our relaxation on y by requiring this restriction only for variables different from y. In particular, we require that if there exists no satisfying assignment tree of φ_x such that the assignments to variables on the left of x in φ are the same in all subtrees for different values assigned to x, then there exists no satisfying assignment tree of φ_x even if we allow y to take different values for different values assigned to x. Thus, by contraposition, our condition in case (ii) can be formulated in terms of assignment trees of φ_x similar as in case (i). Figure 2 illustrates this condition with an arbitrary satisfying assignment tree of φ_x (left), a possible restricted satisfying assignment tree of φ_x (middle), where the values assigned to y are the same in both subtrees for two different values assigned to x, and the corresponding satisfying assignment tree of φ (right).

We are now going to define our observations formally. Let φ be a QCSP formula and $x \in var(\varphi)$. We write φ_x to denote the formula obtained from φ by quantifier reordering such that $\delta_{\varphi_x}(x) = 1$, $\delta_{\varphi_x}(z) = \delta_\varphi(z) + 1$ for all $z \in var(\varphi)$ with $\delta_\varphi(z) < \delta_\varphi(x)$, and $\delta_{\varphi_x}(z) = \delta_\varphi(z)$ for all other $z \in var(\varphi)$. For example, if $\varphi = \exists u \forall v \exists y \forall x \mathcal{C}$, then $\varphi_x = \forall x \exists u \forall v \exists y \mathcal{C}$. The following proposition follows immediately from the

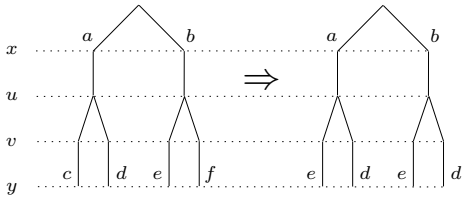


Fig. 1. Independence of y from x in $\varphi = \forall x \, \exists u \, \forall v \, \exists y \, \mathcal{C}$

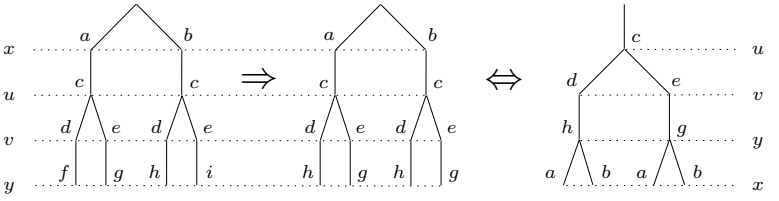


Fig. 2. Independence of y from x in $\varphi = \exists u \, \forall v \, \exists y \, \forall x \, \mathcal{C}$

definition of a satisfying assignment tree (an illustration can be found in the equivalence on the right hand side of Figure 2).

Proposition 1. *Let φ be a QCSP formula and $x \in var_\forall(\varphi)$. Then φ is true if and only if φ_x has a satisfying assignment tree $\mathcal{T}$ such that $\mathcal{T}_{x=d}[z] = \mathcal{T}_{x=d'}[z]$ for all $d, d' \in dom(x)$ and $z \in L_\varphi(x)$.*

Definition 1 (Independence). *Let φ be a QCSP formula and $x \in var_\forall(\varphi)$. A set $Y \subseteq var_\exists(\varphi)$ of variables is* independent *from x if the following holds: If there exists a satisfying assignment tree $\mathcal{T}$ of φ_x such that $\mathcal{T}_{x=d}[z] = \mathcal{T}_{x=d'}[z]$ for all $d, d' \in dom(x)$ and $z \in L_\varphi(x) \setminus Y$, then there exists a satisfying assignment tree $\mathcal{T}'$ of φ_x such that $\mathcal{T}'_{x=d}[z] = \mathcal{T}'_{x=d'}[z]$ for all $d, d' \in dom(x)$ and $z \in Y \cup L_\varphi(x)$.*

Note that we define independence for a set Y of variables instead of individual variables, since the fact that two variables y_1 and y_2 are independent from x does not imply that both variables together, i.e., the set $\{y_1, y_2\}$, is independent from x. For similar reasons it makes sense to use a set of universal variables instead of the single variable x; however, for simplicity, we restrict our considerations in this paper to the case of a single universal variable.

We consider the above notion of independence based on assignment trees fundamental for several applications. In particular, it provides exactly what is needed to characterize those variables that are relevant for clause duplication in Biere's expansion step [4]. Moreover, the above notion also allows us to identify which variables can be shifted within the quantifier prefix as required in backdoor set detection [15]. Note, however, that the other direction does not hold. For example, in the quantifier prefix $\forall x \, \exists u \, \forall y \, \exists v$, variable u may depend on x and y, and variable v may depend on y. Thus, none of the variables can be shifted within the quantifier prefix without affecting the truth value of

C_1	x	u
	2	1
	2	3
	6	1

C_2	y	u	v
	4	3	0
	4	3	5
	7	1	5
	8	1	5

C_3	y	v
	4	5
	7	0
	7	5
	8	5

C_4	u	w
	1	0
	3	4
	3	9

Fig. 3. Example $\psi = \forall x\, \exists u\, \forall y\, \exists v\, \exists w\, \mathcal{C}$, where $\mathcal{C} = \{C_1, C_2, C_3, C_4\}$

the instance; however, v still can be independent from x. For this reason, the following definition of dependency schemes strictly refines the notion introduced in the context of backdoor sets [15].

Definition 2 (Dependency scheme). *A* dependency scheme D *assigns to each QCSP formula* φ*, set* $X \subseteq var_\exists(\varphi)$*, and variable* $x \in var_\forall(\varphi)$ *a set* $D_\varphi(X, x) \subseteq X$ *such that the set* $X \setminus D_\varphi(X, x)$ *of variables is independent from* x*.*

Intuitively, a dependency scheme assigns to each variable x those variables in X whose independence from x is unproven, i.e., all variables in $D_\varphi(X, x)$ are assumed to depend on x. An example of a trivial dependency scheme is $D^{\text{trv}}_\varphi(X, x) := X$ since $X \setminus D^{\text{trv}}_\varphi(X, x) = \emptyset$ is trivially independent from x. Actually, our original motivation for dependency schemes lies in the cases where $X = L_\varphi(x) \cap var_\exists(\varphi)$ and $X = R_\varphi(x) \cap var_\exists(\varphi)$. Because of the more general definition above, however, we do not need to distinguish between these two cases in the remainder of this paper.

Our particular interested lies in *tractable* dependency schemes (i.e., dependency schemes D such that $D_\varphi(X, x)$ can always be computed in polynomial time) that are as general as possible. A dependency scheme D is *more general* than a dependency scheme D' if always $D_\varphi(X, x) \subseteq D'_\varphi(X, x)$ and the inclusion is strict in some cases.

In the following we are going to define two classes of non-trivial tractable dependency schemes. To this aim, we need one more basic definition.

Definition 3 (Connected). *Let* φ *be a QCSP formula with constraint network* $\mathcal{C}$*. An* X*-*path*, $X \subseteq var(\varphi)$*, between two constraints* $C, C' \in \mathcal{C}$ *is a sequence* $C_1, \ldots, C_n$ *of constraints in* $\mathcal{C}$ *with* $C = C_1$ *and* $C' = C_n$ *such that* $var(C_i) \cap var(C_{i+1}) \cap X \neq \emptyset$ *for all* $1 \leq i < n$*. Two constraints* $C, C' \in \mathcal{C}$ *are* connected *with respect to* $X \subseteq var(\varphi)$ *if there is an* X*-path between them.*

For illustration purposes we will consider the following QCSP formula ψ as a running example throughout the remainder of this paper: $\psi = \forall x\, \exists u\, \forall y\, \exists v\, \exists w\, \mathcal{C}$, where $\mathcal{C} = \{C_1, C_2, C_3, C_4\}$ as shown in Figure 3. We assume that the domains consist of the respective values in Figure 3, i.e., $dom(x) = \{2, 6\}$, $dom(y) = \{4, 7, 8\}$, etc. It is easy to see that the constraints C_3 and C_4 are connected with respect to $\{u, v\}$.

3.1 Standard Dependency Scheme

In this section, we consider a dependency scheme that is based on observations of Biere [4] and Bubeck and Kleine Büning [8] in the context of QBF solvers. In particular, Biere [4] noticed that an existential variable y is independent from a universal

variable x if the set of constraints can be partitioned such that x and y are in different partitions and the partitions have no variable in common. This approach of identifying variable dependencies was extended by Bubeck and Kleine Büning [8] in such a way that both partitions may have universal but no existential variables in common. Samer and Szeider [15] incorporated these ideas into their framework of dependency schemes for backdoor set detection. We reformulate the resulting standard dependency scheme for QCSPs based on our refined notion of independence.

Definition 4 (Dependency pair). *Let φ be a QCSP formula with constraint network $\mathcal{C}$ and let $x, y \in var(\varphi)$ such that $q_\varphi(x) \neq q_\varphi(y)$. An* (x, y)-dependency pair *with respect to $X \subseteq var(\varphi)$ is a tuple $C_1, C_2 \in \mathcal{C}$ of constraints such that (i) C_1 and C_2 are connected with respect to X and (ii) $x \in var(C_1)$ and $y \in var(C_2)$.*

In our example ψ from above, there is a (y, w)-dependency pair with respect to $\{u\}$ (by choosing the tuple C_2, C_4) and an (x, u)-dependency pair with respect to $\emptyset$ (by choosing the tuple C_1, C_1).

Definition 5 (Standard dependency scheme). *The* standard dependency scheme D^{std} *assigns to each QCSP formula φ, set $X \subseteq var_\exists(\varphi)$, and variable $x \in var_\forall(\varphi)$ the set $D^{\text{std}}_\varphi(X, x)$ of variables $y \in X$ such that there is an (x, y)-dependency pair with respect to $(R_\varphi(x) \cap var_\exists(\varphi)) \cup X$.*

For instance, recall our running example ψ above and let $X_z = R_\psi(z) \cap var_\exists(\psi)$ for $z \in var_\forall(\psi)$. Then, we have $D^{\text{std}}_\psi(X_x, x) = \{u, v, w\}$ and $D^{\text{std}}_\psi(X_y, y) = \{v\}$. Note that this is an improvement upon the trivial dependency scheme D^{trv}. In fact, the standard dependency scheme is more general than the trivial dependency scheme and it is not hard to show that the difference in size of the sets assigned by D^{std} and D^{trv} can be arbitrarily large.

The following theorem states that the standard dependency scheme is indeed a dependency scheme in our refined sense. This follows immediately from Theorem 2, which will be shown in Section 3.2, since if there is no dependency pair as required in Definition 5, then none of the conditions in Definition 8 can hold.

Theorem 1. *The standard dependency scheme is a dependency scheme.*

The following proposition follows immediately from the fact that traversing the incidence graph of an instance can be performed in linear time.

Proposition 2. *Let φ be a QCSP formula with constraint network $\mathcal{C}$ and let $n = \sum_{C \in \mathcal{C}} |var(C)|$. Then, for every $X \subseteq var_\exists(\varphi)$ and $x \in var_\forall(\varphi)$, the set $D^{\text{std}}_\varphi(X, x)$ can be computed in time $\mathcal{O}(n)$.*

3.2 Triangle Dependency Hierarchy

In the previous section only the structure entailed by the constraint scopes is used in the definition of the standard dependency scheme, i.e., the constraint relations are ignored. In this section we consider a class of dependency schemes that also incorporate the constraint relations, i.e., the possible instantiations satisfying a constraint. The basic

idea behind this kind of dependency scheme was developed in the context of backdoor set detection [15]. In the following, we present a generalization to QCSPs based on our refined notion of independence and, in addition, we generalize it to an infinite hierarchy of dependency schemes such that each dependency scheme is more general than the one on the previous level (in fact, the difference can be arbitrarily large as shown below).

Recall that, in order to show that an existential variable y is independent from a universal variable x, the proof of the standard dependency scheme works by partitioning the set of constraints such that x and y are in different partitions and the partitions have no existential variables in common. The following triangle dependency schemes generalize this idea in such a way that the partitions may have at most one existential variable y in common if y is *pure* in the partition containing x. Gent, Nightingale, and Stergiou [12] introduced the notion of a *pure value* for binary constraints as the CSP counterpart of a pure literal. Our first step is to adapt this notion of pure values to constraints with arbitrary arity.

Definition 6 (Pure value, Pure variable). *Let φ be QCSP formula with constraint network $\mathcal{C}$, $C \in \mathcal{C}$, $x \in var(C)$, and $d \in dom(x)$. We call d a* pure value *of x in C if for all assignments τ on $var(C) \setminus \{x\}$ not falsifying C, $\tau \cup \{(x,d)\}$ satisfies C. We call d a* pure value *of x in $\mathcal{C}' \subseteq \mathcal{C}$ if d is a pure value of x in all $C \in \mathcal{C}'$. Moreover, we call x a* pure variable *in $\mathcal{C}' \subseteq \mathcal{C}$ if there exists a pure value of x in $\mathcal{C}'$.*

For example, recall the QCSP formula $\psi = \forall x \exists u \forall y \exists v \exists w \mathcal{C}$, where $\mathcal{C} = \{C_1, C_2, C_3, C_4\}$ as shown in Figure 3. Here, v is a pure variable in $\{C_2, C_3\}$ since 5 is a pure value of v in both C_2 and C_3. In contrast, u is not pure in C_2 since neither 1 nor 3 are a pure value of u in C_2.

Definition 7 (Dependency triple). *Let φ be a QCSP formula with constraint network $\mathcal{C}$ and let $x, y \in var(\varphi)$ such that $q_\varphi(x) = \forall$ and $q_\varphi(y) = \exists$. An* (x,y)-dependency triple *with respect to $X \subseteq var(\varphi)$ is a triple $C_1, C_2, C_3 \in \mathcal{C}$ of constraints such that (i) C_1 and C_2 as well as C_1 and C_3 are connected with respect to $X \cup \{x\}$, (ii) $x \in var(C_1)$ and $y \in var(C_2) \cap var(C_3)$, and (iii) y is not pure in $\{C_2, C_3\}$.*

In our example ψ from above, there is an (x,w)-dependency triple with respect to $\{u\}$ (by choosing the triple C_1, C_4, C_4) and a (y,u)-dependency triple with respect to $\emptyset$ (by choosing the triple C_2, C_2, C_2).

Roughly speaking, we consider an existential variable y to depend on a universal variable x if a change of the value assigned to x forces a change of the value assigned to y. If the domain of x is at least Boolean (which can be assumed w.l.o.g.) and the domain of y is at most Boolean, this means that y must take each value in its domain; in particular, this also means that y must take a pure value if it has one. This observation is key in the proof of the following triangle dependency hierarchy. In fact, in the definition of the triangle dependency hierarchy we distinguish between Boolean and non-Boolean variables; for non-Boolean variables we use the same ideas as in the standard dependency scheme. Note, however, that Boolean variables that are identified as independent may allow non-Boolean variables to be identified as independent and vice versa, which would not be possible if we consider both cases separately. Moreover, note that variables may become Boolean during the constraint solving process by domain filtering, i.e., by removing domain elements that become known not to be part of a solution.

Definition 8 (Triangle dependency set). *Let φ be a QCSP formula with constraint network $\mathcal{C}$, $X \subseteq var_\exists(\varphi)$, and $x \in var_\forall(\varphi)$. The* triangle dependency set *$\Delta_i(X,x)$ is recursively defined as follows: $\Delta_0(X,x) = (R_\varphi(x) \cap var_\exists(\varphi)) \cup X$ and $\Delta_{i+1}(X,x)$ consists of all $y \in \Delta_i(X,x)$ such that:*

1. *if y is Boolean,*
   (a) *there is an (x,y)-dependency triple with respect to $\Delta_i(X,x) \setminus \{y\}$ or*
   (b) *there exists $z \in \Delta_i(X,x) \cap L^\diamond_\varphi(y)$ such that there is no (x,z)-dependency pair with respect to $\Delta_i(X,x) \setminus \{y\}$, but there is an (x,z)-dependency triple[1] with respect to $\Delta_i(X,x)$;*
2. *otherwise, there is an (x,y)-dependency pair with respect to $\Delta_i(X,x)$.*

Definition 9 (Triangle dependency hierarchy). *The* triangle dependency hierarchy *is a hierarchy of schemes $D^{\triangle_i}$, $i \geq 0$, that assign to each QCSP formula φ, set $X \subseteq var_\exists(\varphi)$, and variable $x \in var_\forall(\varphi)$ the set $D^{\triangle_i}_\varphi(X,x) = \Delta_i(X,x) \cap X$.*

For instance, recall our running example ψ above and let $X_z = R_\psi(z) \cap var_\exists(\psi)$ for $z \in var_\forall(\psi)$. Then, we have $D^{\triangle_1}_\psi(X_x,x) = \{w\}$ and $D^{\triangle_1}_\psi(X_y,y) = \emptyset$, as well as $D^{\triangle_2}_\psi(X_x,x) = D^{\triangle_2}_\psi(X_y,y) = \emptyset$. Note that this is an improvement upon the standard dependency scheme D^{std}. In fact, already the triangle dependency scheme $D^{\triangle_1}$ is more general than the standard dependency scheme, since if there is no (x,y)-dependency pair with respect to $\Delta_0(X,x)$, then none of the conditions in Definition 8 can hold. It is not hard to show that the difference in size of the sets assigned by $D^{\triangle_1}$ and D^{std} can be arbitrarily large. Moreover, note that dependency scheme $D^{\triangle_2}$ tells us that no variable in X_x and X_y depends on x and y, respectively. In particular, this also includes the non-Boolean variable w, which is assumed to depend on x according to the standard dependency scheme.

For each QCSP formula φ, it follows immediately from Definition 8 that the triangle dependency sets must be equal for all $i \geq |var_\exists(\varphi)|$. This observation results in the following definition of the closure of our triangle dependency hierarchy.

Definition 10 (Generalized triangle dependency scheme). *The* generalized triangle dependency scheme *$D^{\triangle_g}$ assigns to each QCSP formula φ, set $X \subseteq var_\exists(\varphi)$, and variable $x \in var_\forall(\varphi)$ the set $D^{\triangle_g}_\varphi(X,x) = \Delta_m(X,x) \cap X$, where m is the smallest number such that $\Delta_m(X,x) = \Delta_{m+1}(X,x)$.*

Let us now consider the relations between the dependency schemes in our triangle dependency hierarchy. We observe that always $\Delta_{i+1}(X,x) \subseteq \Delta_i(X,x)$; thus, it can be easily shown that $D^{\triangle_{i+1}}$ is more general than $D^{\triangle_i}$ for all $i \geq 0$, and that $D^{\triangle_g}$ is more general than $D^{\triangle_i}$ for all $i \geq 0$. The following example shows that already for the special case of QBFs and for any $k \geq 1$, the difference in size of the sets assigned by $D^{\triangle_{k+1}}$ and $D^{\triangle_k}$ and the difference in size of the sets assigned by $D^{\triangle_g}$ and $D^{\triangle_k}$ can be arbitrarily large: Let $n \geq 1$ be an arbitrarily large integer and let $\psi = \forall x\, \exists y_1\, \exists y_2\, \ldots\, \exists y_n\, \exists z_1\, \exists z_2\, \ldots\, \exists z_k\, \mathcal{C}$, where $\mathcal{C}$ consists of the clauses $\{x, z_1, \ldots, z_k\}$, $\{z_k, y_1, \ldots, y_n\}$, $\{z_k, \neg y_1, \ldots, \neg y_n\}$, and $\{z_i, \neg z_{i+1}\}$ for all $1 \leq i < k$. Then, we

[1] If there are no pure variables w.r.t. $\mathcal{C}$, this can be replaced by "(x,z)-dependency pair."

have $D_\psi^{\triangle_k}(X_x, x) = \{y_1, y_2, \ldots, y_n\}$, but $D_\psi^{\triangle_{k+1}}(X_x, x) = D_\psi^{\triangle_g}(X_x, x) = \emptyset$. Thus, the relations between the dependency schemes are as follows:

$$D_\varphi^{\text{trv}}(X, x) \supseteq D_\varphi^{\text{std}}(X, x) \supseteq D_\varphi^{\triangle_1}(X, x) \supseteq D_\varphi^{\triangle_2}(X, x) \supseteq \cdots \supseteq D_\varphi^{\triangle_g}(X, x),$$

where $D_\varphi^{\text{trv}}(X, x) = D_\varphi^{\triangle_0}(X, x)$ and the difference in size between the sets assigned by any two schemes in this hierarchy can be arbitrarily large.

As Condition 1(b) in Definition 8 seems a little unnatural, let us now give an example showing that it cannot be removed: Let $\varphi = \forall x\, \exists z\, \forall u\, \exists y\, \mathcal{C}$, where $\mathcal{C}$ consists of the clauses $\{x, u, \neg y\}, \{\neg x, \neg u, \neg y\}, \{u, y, z\}, \{u, \neg y, \neg z\}, \{\neg u, y, \neg z\}$, and $\{\neg u, \neg y, z\}$. It can be easily checked that φ is true and that there is no (x, y)-dependency triple with respect to $\{z\}$, i.e., y would be considered independent from x if Condition 1(b) was removed. However, there is no satisfying assignment tree $\mathcal{T}$ of φ such that $\mathcal{T}_{x=0}[y] = \mathcal{T}_{x=1}[y]$, i.e., y is not independent from x.

In the following we will show that the triangle dependency hierarchy consists indeed of dependency schemes. Due to space limitations, we only give a proof sketch.

Lemma 1. *Let φ be a QCSP formula with constraint network $\mathcal{C}$ and let $x \in var_\forall(\varphi)$. Moreover, let $X \subseteq var_\exists(\varphi)$ and $Y \subseteq X$. If for all $y \in Y$ there is no (x, y)-dependency pair with respect to X, then the set $\mathcal{C}$ of constraints can be partitioned into two subsets $\mathcal{C}_1$ and $\mathcal{C}_2$ such that (i) $Y \subseteq var(\mathcal{C}_2) \setminus var(\mathcal{C}_1)$, (ii) $x \in var(\mathcal{C}_1) \setminus var(\mathcal{C}_2)$, and (iii) $var(\mathcal{C}_1) \cap var(\mathcal{C}_2) \subseteq var(\varphi) \setminus X$.*

Lemma 2. *Let φ be a QCSP formula with constraint network $\mathcal{C}$ and let $x \in var_\forall(\varphi)$. Moreover, let $X \subseteq var_\exists(\varphi)$ and $Y \subseteq X$. If for all $y \in Y$ there is no (x, y)-dependency triple with respect to $X \setminus \{y\}$, then the set $\mathcal{C}$ of constraints can be partitioned into subsets $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_n$ such that (i) every $y \in Y$ is pure in $\mathcal{C}_1$, (ii) $x \in var(\mathcal{C}_1) \setminus var(\mathcal{C}_i)$ for all $1 < i \leq n$, (iii) $var(\mathcal{C}_1) \cap var(\mathcal{C}_i) \subseteq (var(\varphi) \setminus X) \cup \{y\}$ for all $1 < i \leq n$ and some $y \in Y$, and (iv) $var(\mathcal{C}_i) \cap var(\mathcal{C}_j) \subseteq var(\varphi) \setminus X$ for all $1 < i < j \leq n$.*

Theorem 2. *The triangle dependency hierarchy consists of dependency schemes.*

Proof (Sketch). Let φ be a QCSP formula with constraint network $\mathcal{C}$, $X \subseteq var_\exists(\varphi)$, and $x \in var_\forall(\varphi)$. We show by induction on i that the sets $((R_\varphi(x) \cap var_\exists(\varphi)) \cup X) \setminus \Delta_i(X, x)$ are always independent from x. It follows then immediately that the triangle dependency schemes $D^{\triangle_i}$ are indeed dependency schemes. For the induction start, we have $\Delta_0(X, x) = (R_\varphi(x) \cap var_\exists(\varphi)) \cup X$. Thus, the set $Y_0 = ((R_\varphi(x) \cap var_\exists(\varphi)) \cup X) \setminus \Delta_0(X, x) = \emptyset$ is trivially independent from x. For the induction step, consider the sets $\Delta_k(X, x)$ and $\Delta_{k+1}(X, x)$, and assume that $Y_k = ((R_\varphi(x) \cap var_\exists(\varphi)) \cup X) \setminus \Delta_k(X, x)$ is independent from x. We have to show that $Y_{k+1} = ((R_\varphi(x) \cap var_\exists(\varphi)) \cup X) \setminus \Delta_{k+1}(X, x)$ is also independent from x. To this aim, let $Y = Y_{k+1} \setminus Y_k = \Delta_k(X, x) \setminus \Delta_{k+1}(X, x)$ and $Z = L_\varphi(x) \setminus X$; we partition Y into Y' and Y'' such that Y' consists of all Boolean variables in Y and Y'' consists of all other variables in Y. By Lemma 1 and Lemma 2, this implies that the set $\mathcal{C}$ of constraints can be partitioned into $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_n$ such that (i) every $y \in Y$ is pure in $\mathcal{C}_1$, (ii) $x \in var(\mathcal{C}_1) \setminus var(\mathcal{C}_i)$ for all $1 < i \leq n$, (iii) $var(\mathcal{C}_1) \cap var(\mathcal{C}_i) \subseteq (var(\varphi) \setminus (\Delta_k(X, x) \cup \{x\})) \cup \{y\}$ for all $1 < i \leq n$ and some $y \in Y'$, and

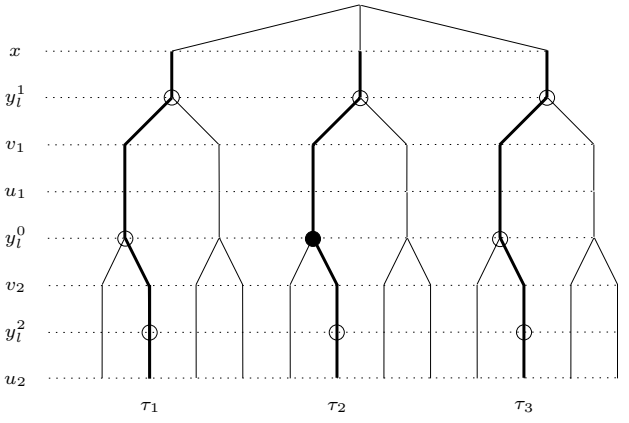


Fig. 4. Illustration of the proof idea of Theorem 2

(iv) $var(\mathcal{C}_i) \cap var(\mathcal{C}_j) \subseteq var(\varphi) \setminus (\Delta_k(X, x) \cup \{x\})$ for all $1 < i < j \leq n$. Now suppose that there exists a satisfying assignment tree $\mathcal{T}$ of φ_x such that $\mathcal{T}_{x=d_1}[z] = \mathcal{T}_{x=d_2}[z]$ for all $d_1, d_2 \in dom(x)$ and $z \in Z$. By induction hypothesis, there exists a satisfying assignment tree $\mathcal{T}'$ of φ_x such that $\mathcal{T}'_{x=d_1}[z] = \mathcal{T}'_{x=d_2}[z]$ for all $d_1, d_2 \in dom(x)$ and $z \in Y_k \cup Z$. We have to show that there exists a satisfying assignment tree $\mathcal{T}''$ of φ_x such that $\mathcal{T}''_{x=d_1}[z] = \mathcal{T}''_{x=d_2}[z]$ for all $d_1, d_2 \in dom(x)$ and $z \in Y_{k+1} \cup Z$. To this aim, we construct $\mathcal{T}''$ by relabeling the vertices of $\mathcal{T}'$ separately for the variables of each partition $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_n$:

Let $\mathcal{C}_l$, $1 < l \leq n$, be any such partition and consider the schematic assignment tree in Figure 4, where we assume that $var(\mathcal{C}_l) \cap Y = \{y_l^0, y_l^1, y_l^2\}$ as depicted by the encircled vertices. Now we choose any path from a child of the root to a leaf; this path corresponds to an assignment, say τ_1. Since every assignment to the universal variables (except x) corresponds uniquely to a path in each subtree below the root, we get the corresponding assignments τ_2 and τ_3. Now, if $var(\mathcal{C}_1) \cap var(\mathcal{C}_l) \cap Y' = \emptyset$ or $y_l^0 \in var(\mathcal{C}_1) \cap var(\mathcal{C}_l) \cap Y'$ is assigned the same value by all three assignments τ_1, τ_2, and τ_3, we can choose any of these assignments. Otherwise, there must be one assignment, say τ_2, which assigns to y_l^0 its pure value as depicted by the filled circle. Thus, we relabel the encircled vertices on the paths corresponding to τ_1 and τ_3 by the labels of the encircled vertices on the path corresponding to τ_2. By repeating this for the other paths in the subtrees and the other partitions appropriately, we are able to construct the required satisfying assignment tree $\mathcal{T}''$. □

We extend the algorithmic idea used for the original triangle dependency scheme [15] to prove the following runtime results.

Proposition 3. *Let φ be a QCSP formula with constraint network $\mathcal{C}$, $m = |var_\exists(\varphi)|$, and $n = \sum_{C \in \mathcal{C}} |var(C)|$. Moreover, let $k \geq 0$ be any fixed integer and suppose that, for every $y \in var_\exists(\varphi)$ and $C \in \mathcal{C}$, we can identify all pure values of y in C in time $\mathcal{O}(\alpha)$.*[2]

[2] If there are no pure variables w.r.t. $\mathcal{C}$, we need the pure values only for Boolean variables.

Then, for every $X \subseteq var_\exists(\varphi)$ and $x \in var_\forall(\varphi)$, the sets $D_\varphi^{\triangle^k}(X,x)$ and $D_\varphi^{\triangle^g}(X,x)$ can be computed in time $\mathcal{O}(n\alpha)$ and $\mathcal{O}(mn\alpha)$, respectively.

Proof. It suffices to prove the runtime for computing $\Delta_{i+1}(X,x)$ from $\Delta_i(X,x)$. To this aim, let $G = (V,E)$ be the incidence graph of an instance, i.e., the bipartite graph with variables and constraints as vertices; a variable x and a constraint C are joined by an edge if and only if $x \in var(C)$. Since $|V| + |E| \leq 3n$, the incidence graph G can be constructed in time $\mathcal{O}(n)$.

Recall that $\Delta_{i+1}(X,x)$ consists of all variables $y \in \Delta_i(X,x)$ satisfying one of the conditions in Definition 8. We check these conditions by performing depth-first search in G starting at x and labeling the visited vertices appropriately; the basic idea behind this approach is due to Tarjan [16]. Note that during the traversal of G, we are allowed to go over all constraints, but only over variables in $\Delta_i(X,x)$, i.e., we have to avoid variables not in $\Delta_i(X,x)$. Let T denote the corresponding search tree, let T_v denote the subtree of T rooted at v, and let $V(T_v)$ denote the set of vertices of T_v. For each vertex $v \in V$, we put $t(v) = j$ if v is the j-th vertex visited. Moreover, we define the low point $lp(v)$ as the least number $t(u)$ over all vertices u in $V(T_v)$ and adjacent (in G) to vertices in $V(T_v) \setminus \{v\}$. It is not hard to see that the labels $t(v)$ and $lp(v)$ can be computed during the top-down and bottom-up phase of a single depth-first traversal, respectively. We also label each variable z with its pure values $pv(z)$ and its temporary pure values $pv'(z)$ as follows: If z is visited the first time, we store in $pv(z)$ the pure values of z in its parent C. Each time we visit a constraint C' adjacent (in G) to z, we put $pv'(z) = pv(z)$ and remove elements from $pv'(z)$ that are not pure values of z in C'. Then, each time we visit z again during the traversal of the subtree rooted at C' via some constraint C'' with $z \in var(C'')$, we remove elements from $pv'(z)$ that are not pure values of z in C''. Finally, after the subtree has been processed and we come back to z, we simply check whether $t(C') < t(z)$ or $lp(C') < t(z)$. If this is the case, we put $pv(z) = pv'(z)$; otherwise, we discard $pv'(z)$. The labels $pv(z)$ can be computed during the same depth-first traversal as above; the pure values of a variable in some constraint can be computed in time $\mathcal{O}(\alpha)$ by assumption. Now we are already able to check Condition 2 and Condition 1(a) in Definition 8: For Condition 2, we just check whether y is reachable from x during our depth-first traversal (which is the case if and only if there is an (x,y)-dependency pair with respect to $\Delta_i(X,x)$), and for Condition 1(a), we check whether y is reachable and $pv(y) = \emptyset$ (which is the case if and only if there is an (x,y)-dependency triple with respect to $\Delta_i(X,x) \setminus \{y\}$). Finally, for Condition 1(b), we label each vertex $v \in V$ with its least-depth non-pure variable $ldn(v)$, i.e., the variable in T_v with least depth in the quantifier prefix that is not pure in the set of all its adjacent constraints, which can be determined in a similar way as for computing $pv(z)$. Again, the labels $ldn(v)$ can be computed during our single traversal of G. Condition 1(b) is then satisfied if and only if y has a child C' such that $lp(C') \geq t(y)$ and $ldn(C')$ belongs to a different quantifier block on the left of the quantifier block y belongs to. Hence, since our depth-first traversal of G can be performed in time $\mathcal{O}(n)$, our runtime results follow. □

An interesting remaining question is about the nature of α in Proposition 3 if we consider a particular constraint language. For example, in the case of extensional binary constraints with finite domain, it is easy to see that α represents the largest domain size

Table 1. Average sizes of the sets assigned by various dependency schemes relative to the sizes of the sets assigned by the trivial dependency scheme and corresponding average runtimes. The sets are computed for all universal variables x over all instances φ with respect to $R_\varphi(x) \cap var_\exists(\varphi)$ without preprocessing (only tautological clauses are removed).

Instances *(# Instances)*	D^{trv}		D^{std}		$D^{\triangle_1}$		$D^{\triangle_2}$		$D^{\triangle_3}$		$D^{\triangle_g}$	
	size (%)	*time* (ms)	*size* (%)	*time* (ms)	*size* (%)	*time* (ms)	*size* (%)	*time* (ms)	*size* (%)	*time* (ms)	*size* (%)	*time* (ms)
QBF Eval'08 (3328)	**100**	0.6	**98.4**	3.1	**31.2**	9.4	**30.1**	11.2	**30.0**	11.5	**29.8**	17.1
QBF Eval'07 (1136)	**100**	0.6	**91.5**	5.8	**72.7**	14.4	**70.6**	20.0	**70.6**	20.0	**70.5**	20.0
QBF Eval'06 (1686)	**100**	0.0	**90.7**	0.4	**48.2**	1.6	**39.4**	2.2	**39.3**	2.9	**39.3**	2.9

squared. In the case of *monotone constraints* [10] with finite domain, deciding whether d is a pure value of y in C reduces to deciding whether a certain assignment satisfies C. Thus, α represents the runtime of checking for each domain element of y whether this assignment satisfies C. Given a linear order on the domain elements, a constraint C is *monotone* if its scope $var(C)$ can be partitioned into S_1 and S_2 such that, for every assignment τ on $var(C)$ that satisfies C, the following holds: If $\tau(y) = a$ and $b \geq a$ for some $y \in S_1$ and $a, b \in dom(y)$, then also $\tau|_{var(C)\setminus\{y\}} \cup \{(y, b)\}$ satisfies C, and, symmetrically, if $\tau(y) = a$ and $b \leq a$ for some $y \in S_2$ and $a, b \in dom(y)$, then also $\tau|_{var(C)\setminus\{y\}} \cup \{(y, b)\}$ satisfies C. We call a QCSP instance *monotone* if all its constraints are monotone. Important subclasses of monotone QCSPs are quantified linear programming (QLP) and QBFs. In the case of QLP with finite domain, α represents the largest domain size times the runtime for evaluating a linear inequation, while in the case of QBFs, α represents the runtime for checking whether $y \in C$ or $\neg y \in C$, which can be done during the traversal of the instance. Hence, for QBFs, the runtime estimations in Proposition 3 become $\mathcal{O}(n)$ and $\mathcal{O}(mn)$, respectively.

In general, if the pure values can be computed in polynomial time, then all schemes in the triangle dependency hierarchy as well as the generalized triangle dependency scheme are tractable by Proposition 3. Thus, for purely theoretical applications, we can always take the generalized triangle dependency scheme since it is more general. However, for practical applications, we can expect that the (practical) runtime increases gradually with the generality of the dependency scheme and thus one has to find a compromise between generality and runtime depending on the actual requirements. Table 1 gives an overview on the average sizes of the sets assigned by various dependency schemes presented in this paper to QBFs from the QBF solver evaluation competitions of the last three years. The sizes of the sets are given relative to the sizes of the sets assigned by the trivial dependency scheme and the runtimes are given in milliseconds.

4 Conclusion

We proposed a refined notion of independence in order to identify variable dependencies caused by the semantics and relative order of the quantifiers in the quantifier prefix of QCSPs. Based on this notion, we redefined the framework of dependency schemes

and presented two classes of concrete dependency schemes, namely the standard dependency scheme and the triangle dependency hierarchy (note that a similar hierarchy construction based on the standard dependency scheme would collapse to its first level). Since our results hold for QCSPs in general, they also hold for important subclasses like QBFs and QLP. In addition to the applications mentioned in the introduction, i.e., expansion-based QBF solvers and the identification of tractable classes of QBFs via backdoor sets, we believe that the knowledge about variable dependencies may also be useful in other related areas. For example, it might be exploited in solvers for other QCSP subclasses and QCSPs in general. In this context, experimental results concerning performance improvements would be interesting. Moreover, we believe that the question of variable dependencies is also interesting from a purely theoretical point of view and that the presented ideas might also be applied in other logic formalisms with quantifiers.

References

1. Ayari, A., Basin, D.: Qubos: Deciding quantified Boolean logic using propositional satisfiability solvers. In: FMCAD 2002. LNCS, vol. 2517, pp. 187–201. Springer, Heidelberg (2002)
2. Benedetti, M.: Quantifier trees for QBFS. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 378–385. Springer, Heidelberg (2005)
3. Benedetti, M.: sKizzo: A suite to evaluate and certify QBFS. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS, vol. 3632, pp. 369–376. Springer, Heidelberg (2005)
4. Biere, A.: Resolve and expand. In: Hoos, H., Mitchell, D.G. (eds.) SAT 2004. LNCS, vol. 3542, pp. 59–70. Springer, Heidelberg (2005)
5. Bordeaux, L., Cadoli, M., Mancini, T.: CSP properties for quantified constraints: Definitions and complexity. In: Proc. 20th National Conference on Artificial Intelligence (AAAI 2005), pp. 360–365. AAAI Press, Menlo Park (2005)
6. Bordeaux, L., Zhang, L.: A solver for quantified Boolean and linear constraints. In: SAC 2007, pp. 321–325. ACM Press, New York (2007)
7. Börner, F., Bulatov, A., Jeavons, P., Krokhin, A.: Quantified constraints: Algorithms and complexity. In: Baaz, M., Makowsky, J.A. (eds.) CSL 2003. LNCS, vol. 2803, pp. 58–70. Springer, Heidelberg (2003)
8. Bubeck, U., Kleine Büning, H.: Bounded universal expansion for preprocessing QBF. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 244–257. Springer, Heidelberg (2007)
9. Chen, H.: Quantified constraint satisfaction and bounded treewidth. In: Proc. 16th European Conference on Artificial Intelligence (ECAI 2004), pp. 161–165. IOS Press, Amsterdam (2004)
10. Dechter, R.: Constraint Processing. Morgan Kaufmann, San Francisco (2003)
11. Egly, U., Tompits, H., Woltran, S.: On quantifier shifting for quantified Boolean formulas. In: Proc. SAT 2002 Workshop on Theory and Applications of Quantified Boolean Formulas, Informal Proceedings, pp. 48–61 (2002)
12. Gent, I.P., Nightingale, P., Stergiou, K.: QCSP-Solve: A solver for quantified constraint satisfaction problems. In: Proc. 19th International Joint Conference on Artificial Intelligence (IJCAI 2005), pp. 138–143. Professional Book Center (2005)

13. Gottlob, G., Greco, G., Scarcello, F.: The complexity of quantified constraint satisfaction problems under structural restrictions. In: Proc. 19th International Joint Conference on Artificial Intelligence (IJCAI 2005), pp. 150–155. Professional Book Center (2005)
14. Lonsing, F., Biere, A.: Nenofex: Expanding NNF for QBF solving. In: Kleine Büning, H., Zhao, X. (eds.) SAT 2008. LNCS, vol. 4996, pp. 196–210. Springer, Heidelberg (2008)
15. Samer, M., Szeider, S.: Backdoor sets of quantified boolean formulas. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 230–243. Springer, Heidelberg (2007)
16. Tarjan, R.E.: Depth first search and linear graph algorithms. SIAM Journal of Computing 1(2), 146–160 (1972)

Treewidth: A Useful Marker of Empirical Hardness in Quantified Boolean Logic Encodings

Luca Pulina and Armando Tacchella*

DIST, Università di Genova, Viale Causa, 13 – 16145 Genova, Italy
Luca.Pulina@unige.it, Armando.Tacchella@unige.it

Abstract. Theoretical studies show that in some combinatorial problems, there is a close relationship between classes of tractable instances and the treewidth (tw) of graphs describing their structure. In the case of satisfiability for quantified Boolean formulas (QBFs), tractable classes can be related to a generalization of treewidth, that we call quantified treewidth (tw_p). In this paper we investigate the practical relevance of computing tw_p for problem domains encoded as QBFs. We show that an approximation of tw_p is a predictor of empirical hardness, and that it is the only parameter among several other candidates which succeeds consistently in being so. We also provide evidence that QBF solvers benefit from a preprocessing phase geared towards reducing tw_p, and that such phase is a potential enabler for the solution of hard QBF encodings.

1 Introduction

Several theoretical studies deal with the relationship between the complexity of combinatorial problems and the treewidth (tw) of graphs representing their structure. The common trait of such studies is that the assumption of bounded values of tw yields tractable classes of problems which are intractable otherwise. This connection has been unveiled in the study of graph algorithms, and it emerged in other areas of application (see, e.g. [1]). In the context of the constraint satisfaction problem (CSP), the connection was first explored by Freuder [2], Dechter and Pearl [3], and more recent results (see, e.g., [4,5,6]) consider also the quantified constraint satisfaction problem (QCSP).

In this paper, we are concerned with the practical relevance of the above results for problems that can be encoded as quantified Boolean formulas (QBFs). The satisfiability problem for QBFs (QSAT) is the subclass of QCSP wherein all the variables range over a Boolean domain. The importance of QSAT stems both from theoretical aspects – QSAT is the prototypical PSPACE-complete problem [7] – and from the fact that QBFs can provide compact Boolean encodings in several automated reasoning tasks. The interest in QSAT is also witnessed by a number of QBF encodings and solvers (see [8]), and by the presence of an annual competition of QBF solvers (QBFEVAL) [9].

From a practical standpoint, we see the results in [4,5,6] as gateways to the efficient solution of QBF encodings – many of which are also industrially relevant. In particular, we build on [4] which relates the complexity of solving QCSPs to a generalization of tw that we call quantified treewidth (tw_p). Our main contributions, obtained considering data from the three most recent QBFEVAL events (2006-2008), are the following:

* This work has been partially supported by the Italian Ministry of University and Research.

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 528–542, 2008.

- Since computing tw is an NP-complete problem [10] it is not difficult to see that tw_p must be NP-hard at least; however, it turns out that tw_p can be *approximated* efficiently enough; to this purpose we introduce the proof-of-concept tool QuTE, a **Qu**antifed **T**reewidth **E**stimator, to compute upper bounds of tw_p.
- While bounding tw_p is only a *sufficient* condition for tractability and we have no clue whether increasing it will correspond to an increase in difficulty, we show that the approximation of tw_p computed by QuTE is a robust predictor – albeit in a statistical sense – of the performances exhibited by solvers when coping with QBF encodings; in this sense, the approximation of tw_p is a marker of empirical hardness, and it is the only parameter that succeeds consistently in being so among several other syntactic parameters which are plausible candidates.
- The result in [4] relates to a specific algorithm and it does not say much about QBF solvers using different approaches like search (see, e.g., [11,12]), skolemization (see, e.g., [13]), or variable elimination (see, e.g., [14,15]);[1] however, our experimental analysis shows that the significance of approximated tw_p is not related to some specific solver only.
- Finally, computing approximations of tw_p is also useful; to show this we introduce QuBIS, a **Qu**antified **B**oolean formula **I**ncomplete **S**olver. QuBIS is incomplete in that, given an input QBF φ, it may either solve φ, or halt producing another QBF φ' whose treewidth is no larger than the treewidth of φ in most cases. Experiments with QuBIS show that preprocessing helps when it decreases the treewidth of QBFs, and the improvement can be so dramatic that formulas which cannot be solved by any solver before QuBIS preprocessing, can be solved afterwards.

The paper is structured as follows. In Section 2 we introduce basic definitions. In Section 3 we introduce the concept of empirical hardness and we show that the approximation of tw_p provided by QuTE is a marker of solver performances. In Section 4 we describe QuBIS in some detail, and show that it can effectively improve the performances of state-of-the-art QBF solvers. We conclude the paper in Section 5 with a summary about our current results and the related work.

2 Preliminaries

In this section we consider the definition of QBFs and their satisfiability as given in the literature of QBF decision procedures (see, e.g., [11,13,14]), and we introduce notation from [4] to define graphs and associated parameters describing the structure of QBFs.

A *variable* is an element of a set P of propositional letters and a *literal* is a variable or the negation thereof. We denote with $|l|$ the variable occurring in the literal l, and with $\overline{l}$ the *complement* of l, i.e., $\neg l$ if l is a variable and $|l|$ otherwise. A literal is *positive* if $|l| = l$ and *negative* otherwise. A *clause* C is an n-ary ($n \geq 0$) disjunction of literals such that, for any two distinct disjuncts l, l' in C, it is not the case that $|l| = |l'|$. A

[1] As shown in [16], the performances of Boolean satisfiability solvers based on variable elimination are very sensitive to different values of tw. As we confirm in Section 4, QBF solvers based on variable elimination are those that are most sensitive to tw_p and thus most closely resemble an implementation of the algorithm used in [4].

propositional formula is a k-ary ($k \geq 0$) conjunction of clauses. A *quantified Boolean formula* is an expression of the form

$$Q_1 z_1 \ldots Q_n z_n \Phi \tag{1}$$

where, for each $1 \leq i \leq n$, z_i is a variable, Q_i is either an existential quantifier $Q_i = \exists$ or a universal one $Q_i = \forall$, and Φ is a propositional formula in the variables $\{z_1, \ldots, z_n\}$. The expression $Q_1 z_1 \ldots Q_n z_n$ is the *prefix* and Φ is the *matrix* of (1). A literal l is *existential* if $|l| = z_i$ for some $1 \leq i \leq n$ and $\exists z_i$ belongs to the prefix of (1), and it is *universal* otherwise. For example, the following expression is a QBF:

$$\begin{aligned}\forall y_1 \exists x_1 \forall y_2 \exists x_2 \exists x_3 ((y_1 \vee y_2 \vee x_2) \wedge (y_1 \vee \neg y_2 \vee \neg x_2 \vee \neg x_3) \wedge \\ (y_1 \vee \neg x_2 \vee x_3) \wedge (\neg y_1 \vee x_1 \vee x_3) \wedge \\ (\neg y_1 \vee y_2 \vee x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2) \wedge \\ (\neg y_1 \vee \neg x_1 \vee \neg y_2 \vee \neg x_3) \wedge \\ (\neg x_2 \vee \neg x_3)).\end{aligned} \tag{2}$$

The semantics of a QBF φ can be defined recursively as follows. A QBF clause is *contradictory* exactly when it does not contain existential literals. If the matrix of φ contains a contradictory clause then φ is false. If the matrix of φ has no conjuncts then φ is true. If $\varphi = Qz\psi$ is a QBF and l is a literal, we define φ_l as the QBF obtained from ψ by removing all the conjuncts in which l occurs and removing $\overline{l}$ from the others. Then we have two cases. If φ is $\exists z\psi$, then φ is true exactly when φ_z or $\varphi_{\neg z}$ are true. If φ is $\forall z\psi$, then φ is true exactly when φ_z and $\varphi_{\neg z}$ are true. The QBF satisfiability problem (QSAT) is to decide whether a given formula is true or false. It is easy to see that if φ is a QBF without universal quantifiers, solving QSAT is the same as solving propositional satisfiability (SAT).

A *relational signature* σ is a finite set of relation symbols, each of which has an associated arity. A (finite) *relational structure* $\mathbf{A}$ over σ consists of a universe A and a relation $R^{\mathbf{A}}$ over A for each relation symbol R of σ, such that the arity of $R^{\mathbf{A}}$ matches the arity associated to R. Accordingly, the QBF (2) can be rewritten as:

$$\begin{aligned}\forall y_1 \exists x_1 \forall y_2 \exists x_2 \exists x_3 (C_{000}(y_1, y_2, x_2) \wedge C_{0111}(y_1, y_2, x_2, x_3) \wedge \\ C_{010}(y_1, x_2, x_3) \wedge C_{100}(y_1, x_1, x_3) \wedge \\ C_{100}(y_1, y_2, x_2) \wedge C_{101}(y_1, y_2, x_2) \wedge \\ C_{1111}(y_1, x_1, y_2, x_3) \wedge C_{11}(x_2, x_3))\end{aligned} \tag{3}$$

over the signature $\sigma = \{C_{000}, C_{0111}, C_{010}, C_{100}, C_{101}, C_{1111}, C_{11}\}$ where each $C_w \in \sigma$ has arity $|w|$. Let ϕ be the expression (3). The QSAT problem for ϕ can be restated as the problem of checking the (first-order logic) entailment $\mathbf{B} \models \phi$, where $\mathbf{B}$ is a relational structure with signature σ and universe $B = \{0, 1\}$, such that, for each $C_w \in \sigma$, $C_w^{\mathbf{B}}$ is the relation containing all $|w|$-tuples over B except w, e.g., $C_{11}^{\mathbf{B}} = \{(0, 0), (0, 1), (1, 0)\}$.

Following [4] we further introduce the notion of *quantified relational structure* as a pair $(p, \mathbf{A})$ where $\mathbf{A}$ is a relational structure and p is a prefix, i.e., an expression of the form $Q_1 z_1 \ldots Q_n z_n$ where each Q_i is either $\exists$ or $\forall$, and $z_1, \ldots, z_n$ are exactly the elements of the universe of $\mathbf{A}$. The quantified relational structure $(p, \mathbf{A})$ associated to

a QBF ϕ is obtained by letting p be the prefix of ϕ and letting $R^{\mathbf{A}}$ contain all tuples $(a_1, \ldots, a_k)$ such that $R(a_1, \ldots, a_k)$ appears as a conjunct in ϕ. Notice that a prefix $p = Q_1 z_1 \ldots Q_n z_n$ can be viewed as the concatenation of *quantifier blocks* where quantifiers in each block are the same, and consecutive blocks have different quantifiers. If $h \leq n$ is the number of blocks in p, then $h-1$ is the *alternation depth* of p and, by extension, of the QBF having p as a prefix. If p consists of the blocks $Q_1 Z_1 \ldots Q_h Z_h$, then to each variable z we can associate a *level* $l(z)$ which is the index of the corresponding block, i.e., $l(z) = i$ for all the variables $z \in Z_i$. We also say that variable z_1 *comes after* a variable z_2 in p if $l(z_1) \geq l(z_2)$. For instance, the quantified relational structure associated to (3) is $(\forall y_1 \exists x_1 \forall y_2 \exists x_2 \exists x_3, \mathbf{A})$, with universe $A = \{y_1, y_2, x_1, x_2, x_3\}$ and

$$\begin{array}{ll} C^{\mathbf{A}}_{000} = \{C_{000}(y_1, y_2, x_2)\} & C^{\mathbf{A}}_{0111} = \{C_{0111}(y_1, y_2, x_2, x_3)\} \\ C^{\mathbf{A}}_{010} = \{C_{010}(y_1, x_2, x_3)\} & C^{\mathbf{A}}_{100} = \{C_{100}(y_1, x_1, x_3), C_{100}(y_1, y_2, x_2)\} \\ C^{\mathbf{A}}_{1111} = \{C_{1111}(y_1, x_1, y_2, x_3)\} & C^{\mathbf{A}}_{11} = \{C_{11}(x_2, x_3)\}. \end{array} \quad (4)$$

A relational structure – and thus the structure of a QBF – can be described by a *Gaifman graph*. Given a relational structure $\mathbf{A}$, the Gaifman graph of $\mathbf{A}$ is the graph with vertex set equal to the universe A of $\mathbf{A}$ and with an edge (a, a') for every pair of different elements $a, a' \in A$ that occur together in some $\mathbf{A}$-tuple, i.e., in some element of $R^{\mathbf{A}}$ for some relation symbol R. A *scheme* for a quantified relational structure $(p, \mathbf{A})$ is a supergraph (A, E) of the Gaifman graph of $\mathbf{A}$ along with and ordering $a_1, \ldots, a_n$ of the elements of A such that

1. the ordering $a_1, \ldots, a_n$ preserves the order of p, i.e., if $i < j$ then a_j comes after a_i in p, and
2. for any a_k, its lower numbered neighbors form a clique, that is, for all k, if $i < k$, $j < k$, $(a_i, a_k) \in E$ and $(a_j, a_k) \in E$, then $(a_i, a_j) \in E$

The *width* w_p of a scheme is the maximum, over all vertices a_k, of the size of the set $\{i : i < k, (a_i, a_k) \in E\}$, i.e., the set containing all lower numbered neighbors of a_k. The *treewidth* tw_p of a quantified relational structure $(p, \mathbf{A})$ is the minimum width over all schemes for $(p, \mathbf{A})$. Given the correspondence between relational structures and QBFs, we write $tw_p(\varphi)$ to denote the treewidth of the QBF φ.

3 Treewidth and Empirical Hardness

If we define the class QSAT[$tw_p < k$] as the restriction of the QSAT problem to all instances $((p, \mathbf{A}), \mathbf{B})$ where $(p, \mathbf{A})$ has quantified treewidth strictly less than k, then, considering the definition of the polynomial-time algorithm k-consistency of [4], we can state the following:

Theorem 1 ([4]). *For all $k \geq 2$, establishing k-consistency is a decision procedure for QSAT[$tw_p < k$]*

From the above, it immediately follows that the class QSAT[$tw_p < k$] is a tractable subclass of the QSAT problem and, therefore, the QBFs corresponding to relational structures with a bounded tw_p are a tractable subclass of QSAT.

```
SORTBYPREFIX(φ, {v_1, v_2, ..., v_n})
 1 h ← number of quantifier blocks of φ
 2 Z_h ← h-th quantifier block of φ
 3 i ← 1; j ← 1; s ← 0
 4 while (h > 0) do
 5   if (v_i ∈ Z_h) then
 6     v'_j ← v_i; j ← j + 1; Z_h ← Z_h \ {v_i}
 7     if (Z_h = ∅) then
 8       h ← h - 1
 9       if (s ≠ 0) then i ← s; s ← 0
10   else
11     if (s = 0) then s ← i
12     while (v_i ∉ Z_h) i ← i + 1
13 return {v'_1, v'_2, ... v'_n}
```

```
FILLIN ((V, E), {v_1, v_2, ..., v_n})
 1 tw ← 0
 2 for v ∈ V do
 3   M(v) ← the set of vertices adjacent to v
 4 E' ← E
 5 for i ← 1 to n do
 6   if |M(v_i)| > tw then
 7     tw ← |M(v_i)|
 8   for u, w ∈ M(v_i) and (u, w) ∉ E' do
 9     E' ← E' ∪ (u, w)
10     M(u) ← M(u) ∪ {w}
11     M(w) ← M(w) ∪ {u}
12   for u ∈ M(v_i) do
13     M(u) ← M(u)\{v_i}
14 return tw
```

```
QUTE(φ)
 1 G ← Gaifman graph of φ
 2 σ ← FINDORDERING(G)
 3 σ' ← SORTBYPREFIX(φ, σ)
 4 t̂w_p ← FILLIN(G, σ')
 5 return t̂w_p
```

Fig. 1. The algorithm of QUTE and the auxiliary functions SORTBYPREFIX and FILLIN

To understand the implications of Theorem 1 in the case of concrete QBF encodings and solvers, we define a notion of empirical hardness. Given a set of QBFs Γ, a set of QBF solvers Σ, and an implementation platform Π, we define *hardness* as a partial function $H_{\Gamma,\Sigma,\Pi} : \Gamma \to \{0, 1\}$ such that

- $H_{\Gamma,\Sigma,\Pi}(\varphi) = 1$ iff no solver in Σ can solve φ on Π, and
- $H_{\Gamma,\Sigma,\Pi}(\varphi) = 0$ iff all solvers in Σ can solve φ on Π.

The parameters Γ and Σ take into account the dependency of H from the current state of the art in the available QBF encodings and solvers. The parameter Π denotes the dependency of H from the currently available hardware platforms, as well as the amount of time and space resources allotted to the solvers. In the remainder of this section Γ, Σ and Π are fixed, and we write $H(\varphi)$ to denote the hardness of φ. In particular, Π is a family of identical Linux workstations comprised of 8 Intel Core 2 Duo 2.13 GHz PCs with 4GB of RAM; the resources granted to the solvers are 600s of CPU time and 3GB of memory. As for Γ and Σ, we consider the sets of formulas and solvers that participated in the three most recent competitions of QBF solvers (QBFEVAL) [9].

Understanding the relationship between $H(\varphi)$ and tw_p requires the computation of both quantities for all the formulas in Γ, but computing an exact value of tw_p is unfeasible with our choice of Γ. Therefore, we built the tool QUTE – whose implementation is sketched in Figure 1 – in order to provide us with approximate values of tw_p. As we can see in Figure 1, QUTE takes as input a QBF φ and returns an approximate value $\hat{tw}_p$ of the quantified treewidth by performing the following steps:

- The input QBF is converted to the corresponding Gaifman graph G (line 1) according to the construction presented in Section 2.
- An ordering σ is sought (line 2); the current implementation of FINDORDERING is the maximum cardinality search (MCS) algorithm of [17].
- The function SORTBYPREFIX (line 3) transforms σ into another – possibly identical – ordering σ' which is compatible with the prefix of φ.
- Finally, the function FILLIN (line 4) computes the value of $\hat{tw}_p$ by computing a chordal completion of G in such a way that σ' becomes a perfect elimination scheme; since σ' is not guaranteed to yield the minimum value of tw over all possible chordal completions, it turns out that $\hat{tw}_p \geq tw_p$.

There are two important observations about QUTE. First, if FINDORDERING were able to guess σ in such a way that the chordalization performed by FILLIN yields the minimum maximal clique over all possible chordal completions, then $\hat{tw}_p = tw_p$. In practice, FINDORDERING is just an heuristic, but efficient heuristics – like MCS – do not guarantee a tight bound on the approximation, while more accurate algorithms (like, e.g., QuickBB [18]) are hopelessly slow in our case. We have experimented with several FINDORDERING – indeed, all those available in the TreeD library [19] on top of which QUTE is implemented – and we did not find substantial differences among different heuristics. Also, for graphs in which tw_p can be computed – random graphs of up to 30 nodes – we have seen that decreasing tw_p causes also $\hat{tw}_p$ to decrease and, if the graph is either very sparse of mostly connected, $tw_p = \hat{tw}_p$ on most samples. However, the question whether $\hat{tw}_p$ is a tight approximation of tw_p remains open for the kind of graphs that we deal with. Second, all the QBFs that we consider are in prenex CNF, meaning that all the bound variables are constrained to a total order. However, it has been shown (see, e.g., [20,21]) that the "true" prefix structure is, in many cases, a partial order among the bound variables. Using partial orders instead of total ones may allow, in some cases, to obtain a better approximation of tw_p than the one computed starting from the prenex QBF. We consider all the above issues related to improving QUTE[2] as topics for future research.

The main result of this section is presented in Figure 2 where we consider *hard* the formulas φ such that $H(\varphi) = 1$, and *easy* the formulas φ such that $H(\varphi) = 0$. For each QBFEVAL dataset, in Figure 2 (left) we show plots obtained by considering each formula as a point in the multidimensional space of *syntactic features*, a set of 141 parameters that can be computed inexpensively like, e.g., the number of clauses, the number of variables and the alternation depth. The complete listing and a detailed description of such parameters can be found in [23]. Since it is impossible to visualize a space in 141 dimensions, we consider its two-dimensional projection obtained by means of a principal components analysis (PCA) and considering only the first two principal components.[3] In Figure 2 (right) we show plots obtained by considering the distributions of $\hat{tw}_p$ computed for easy and hard formulas by QUTE. For each distribution, we show a box-and-whiskers diagram representing the median (bold line), the first and third quartile (bottom and top edges of the box), the minimum and maximum (whiskers at the

[2] The latest C++ implementation of QUTE is available at [22].

[3] Details about PCA and its use for visualizing multidimensional datasets are beyond the scope of this paper: see, e.g., Chap. 7 of [24] for an introduction to PCA and further references.

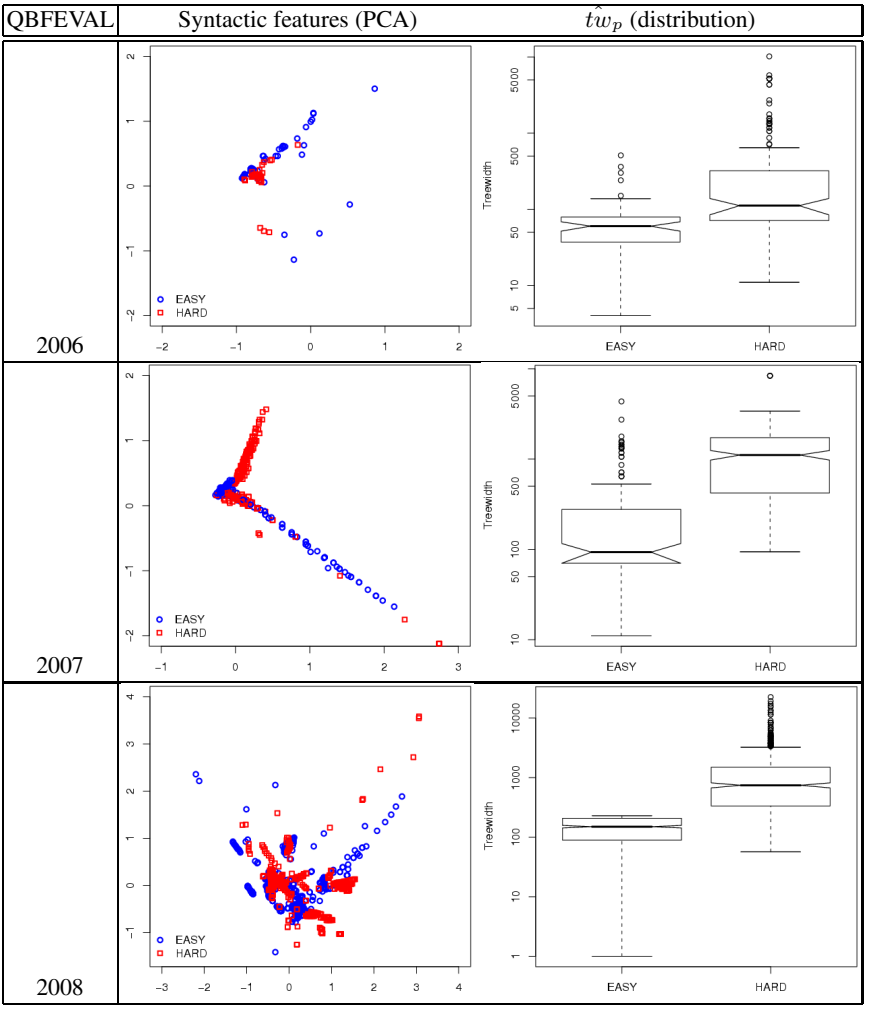


Fig. 2. Hardness vs. syntactic features (left) and treewidth (right)

top and the bottom) of a distribution. Values laying farther away than the median ± 1.5 times the interquartile range are considered outliers and shown as dots on the plot.[4] An approximated 95% confidence interval for the difference in two medians is represented by the notches cut in the boxes: if the notches of two plots do not overlap, this is strong evidence that the two medians differ.

Considering the results shown in Figure 2 (right) we can conclude that $\hat{tw}_p$ is a robust marker of empirical hardness, since for all QBFEVAL datasets the distribution of

[4] In case outliers are detected, the whiskers extend up to the median $+1.5$ (resp. -1.5) times the interquartile range, while the maximum (resp. minimum) value becomes the highest (resp. lowest) outlier.

Table 1. Discriminative power of syntactic features considering posterior probabilities

Group	Feature	Accuracy (%)
Treewidth ($\hat{tw}_p$)		72
Number of variables	Existential	49
	Universal	64
	Total	49
Number of sets	Total	54
Number of variables per set	Existential	45
	Universal	62
	Total	46
Clauses-to-Variables		43

Group	Feature	Accuracy (%)
Number of clauses	Unary	42
	Binary	53
	Horn	50
	Dual Horn	50
	Total	58
Number of occurrences per variable (mean)	Existential	59
	Universal	44
	Negative	42
	Positive	41
	Total	41

$\hat{tw}_p$ varies significantly across hard and easy instances. To get a quantitative feeling of this, let us consider a simple Bayesian argument related to the problem of deciding whether a given QBF φ is hard or not. For the sake of our argument, the distributions in Figure 2 (right) represent $p(\hat{tw}_p|H)$, i.e., the probability density of $\hat{tw}_p$ given the hardness H. The proportion of hard instances in each dataset is the parameter r in $p(H) = r^H(1-r)^{(1-H)}$, i.e., the prior density of hardness H. Looking at prior information only, we decide that a QBF φ is hard exactly when $P(H = 1) > P(H = 0)$, i.e., when $r > 0.5$. If we consider also the likelihood $p(\hat{tw}_p|H)$, then we can compute the posterior density $p(H|\hat{tw}_p)$ using Bayes' rule and then decide that a QBF φ is hard exactly when

$$P(\hat{tw}_p(\varphi)|H(\varphi) = 1) \cdot P(H(\varphi) = 1) > P(\hat{tw}_p(\varphi)|H(\varphi) = 0) \cdot P(H(\varphi) = 0)$$

We close our argument by stating that in all the QBFEVAL datasets the accuracy of the above criterion is strictly higher than looking at prior probabilities only. For instance, in the QBFEVAL'08 dataset we have that $r = 0.54$ meaning that $P(H = 0) = 0.46$ and $P(H = 1) = 0.54$. A priori, given a formula φ in the QBFEVAL'08 dataset we would decide that it is hard ($H = 1$), with a 54% accuracy, i.e., only slightly more than tossing a fair coin. With the Bayesian approach, looking at $\hat{tw}_p$ we obtain a 72% accuracy which is a definite increase in our predictive ability.

On the other hand, syntactic features are collectively unable to distinguish among easy and hard instances. As we can see in Figure 2 (left), with the partial exception of some hard instances in the QBFEVAL'07 dataset, there is a non-negligible chance that easy and hard instances share similar values of such features. Technically, we say that in the space of syntactic features it is hard to find a discriminant – a curve in the PCA plots of Figure 2 – that allows us to tell easy instances from hard ones with sufficient accuracy. Repeating the Bayesian argument above for syntactic features on the QBFEVAL'08 dataset, we obtain the results of Table 1. Here we can see that posterior probability densities do not yield substantial improvements over prior probabilities except in the case of $\hat{tw}_p$ meaning that, overall, syntactic features are far from the predictive power of approximated treewidth. In some cases, including the "famous" clauses-to-variable ratio, conditioning hardness on a syntactic feature is even worse than tossing a fair coin.

4 Treewidth and Useful Preprocessing

In this section we introduce our tool QuBIS, an incomplete solver which, given an input QBF φ, may either solve it, or halt producing another QBF φ'. With a suitable setting of the parameters of QuBIS, $\hat{tw}_p(\varphi')$ can be made no larger than $\hat{tw}_p(\varphi)$ in most cases. The purpose of this section is to show that QBF solvers benefit from getting the output of an incomplete run of QuBIS rather than being fed the original QBF as input. In other words, preprocessing helps when it decreases $\hat{tw}_p$ and, in practice, this is often true independently from the algorithm featured by the solver.

QuBIS is based on *Q-resolution* defined in [25] as an operation among clauses of a QBF. In particular, given two clauses $P \vee x$ and $R \vee \neg x$, where P and R are disjunctions of literals, the clause $P \vee R$ can be derived by Q-resolution subject to the constraints that (i) x is an existential variable, and (ii) P and R do not share any variable z such that $\neg z$ (resp. z) occurs in P and z (resp. $\neg z$) occurs in R. QuBIS uses Q-resolution to perform *variable elimination* on existential variables defined, e.g., in [14], as the operation whereby, given a QBF $Q_1 z_1 Q_2 z_2 \ldots \exists x \Phi$, the variable x can be resolved away by performing all resolutions on x, adding the resolvents to the matrix Φ and removing from Φ all the clauses containing x. Universal variables can be eliminated simply by deleting them once they have the highest prefix level. More precisely, given a matrix Φ, let $\Phi_{/z}$ be the same matrix whereby all the occurrences of z have been deleted. The QBF $Q_1 z_1 Q_2 z_2 \ldots \forall y \Phi$ is true exactly when the QBF $Q_1 z_1 Q_2 z_2 \ldots \Phi_{/y}$ is true, so y can be eliminated safely. From the above, it immediately follows that variable elimination, once it respects the prefix order, yields a decision procedure for QSAT (see, e.g., [15,14]).

QuBIS takes as input a QBF φ and two parameters: (i) an integer deg, the maximum degree allowed for a given variable considering the Gaifman graph of φ; (ii) an integer div, the maximum value of *diversity*, a parameter defined in [16] as the product of the number of positive and negative occurrences of a variable in φ. The role of deg is thus to bound the number of variables in a clause, while the role of div is to bound the (worst case) number of resolvents generated when eliminating an existential variable. Intuitively, QuBIS eliminates variables until the input QBF can be declared true, false or when eliminating variables is bound to increase the size of the resulting QBF beyond some critical threshold. More precisely, a variable qualifies for elimination only if it has the highest level in the prefix of φ, and

- it is a universal variable, or
- it is an existential variable, its degree is no larger than deg and its diversity is no larger than div.

Universal variables are eliminated simply by deleting all their occurrences from the matrix of φ, while existential variables are resolved away. In both cases, the resulting QBF is given as argument to a recursive call of QuBIS. QuBIS terminates when one of the following conditions is satisfied: (i) the matrix of φ is empty – in which case the input QBF is true; (ii) the matrix of φ contains a contradictory clause – in which case the input QBF is false; (iii) there are no variables that qualify for elimination in φ – in which case φ is returned as output. Therefore, QuBIS is a sound and complete decision

Table 2. Encodings to experiment with QuBIS(top), and performances of QBF solvers (bottom)

Encoding	QBFs	Description
add	32	equivalence checking of partial implementations of circuits
circ	63	FPGA logic synthesis
count	24	model checking of counter circuits
cp	24	conformant planning domains
k	378	modal K formulas
katz	20	symbolic reachability of industrially relevant circuits
s	171	symbolic diameter evaluation of ISCAS89 circuits
tipdiam	203	symbolic diameter evaluation of circuits

Solver	add		circ		count		cp		k		katz		s		tipdiam	
	#	Time	#	Time	#	Time	#	Time	#	Time	#	Time	#	Time	#	Time
QMRes	20	1061.53	–	–	8	87.53	1	0.30	269	3072.22	7	51.90	6	36.00	58	2576.02
Quantor	8	20.94	7	141.66	12	16.05	16	1710.01	259	922.56	–	–	17	1185.52	76	709.27
QuBE3.0	5	1.99	4	18.08	9	90.15	6	160.08	115	5552.56	–	–	1	0.07	71	1132.97
QuBE6.1	5	1.34	4	4.87	9	116.52	5	169.14	203	4376.43	6	26.21	62	3006.31	151	1493.88
sKizzo	14	814.80	6	79.11	12	5.88	7	287.40	348	7262.20	–	–	19	1089.13	133	8554.86
yQuaffle	4	0.86	4	0.58	9	3.99	8	267.22	142	5622.60	–	–	1	0.12	71	2162.84

procedure for the subclass of QBFs in which variables always qualify for elimination, while for all the other formulas QuBIS[5] behaves like a preprocessor.

The experiments detailed in this section are carried out on the same computing platforms described in Section 3, but here we focus on the 915 QBF encodings summarized in Table 2 (top) and on the following QBF solvers (references available from [8]):

QMRes is a symbolic implementation of variable elimination featuring multi-resolution, unit propagation, and heuristics to choose variables.

Quantor uses existential variable elimination and universal variables expansion, plus equivalence reasoning, subsumption checking, pure and unit literal detection.

QuBE3.0 is a search-based solver with learning.

QuBE6.1 is a composition of the search-based solver QuBE and a preprocessor that applies equivalence substitution, Q-resolution and clause subsumption.

sKizzo is a reasoning engine for QBF featuring several techniques, including search, resolution and skolemization.[6]

yQuaffle is a search-based solver featuring learning and inversion of quantifiers.

Our first experiment is to run the solvers on the QBF encodings, with the goal of showing that the encodings are challenging enough given the current state of the art, and that the algorithms featured by the solvers are "orthogonal", i.e., solvers have complementary abilities across different families. Table 2 (bottom) shows the results: the first column contains the solver names, and it is followed by eight groups of columns, one for each encoding. The columns "#" and "Time" contain, respectively, the number of formulas solved and the cumulative CPU seconds. A dash on both columns means that the solver did not solve any formula. Looking at Table 2 (bottom) we see that our selection is indeed valid for our purposes. For instance, considering circ encodings we see that no single solver is able to solve more than about 10% of them. Still considering the percentage of QBFs solved by any single solver, we see that a similar result holds

[5] A proof-of-concept implementation in C++ of QuBIS can be downloaded from [22].

[6] sKizzo is run with its default settings.

Table 3. Results of treewidth analysis on QBF encodings and their preprocessed versions

Solver	add						circ						count						cp					
	H	H'	H'_q	$<$	μ	μ_q	H	H'	H'_q	$<$	μ	μ_q	H	H'	H'_q	$<$	μ	μ_q	H	H'	H'_q	$<$	μ	μ_q
QMRES	12	12	9	9	996	705	63	54	35	35	2055	1865	16	16	16	4	253	232	23	21	18	18	173	163
QUANTOR	24	24	21	21	541	398	56	47	28	28	2466	2234	12	12	12	3	311	309	8	6	4	4	447	438
QUBE3.0	27	27	24	23	507	364	59	50	31	31	2304	2090	15	15	15	4	254	229	18	16	13	13	211	203
QUBE6.1	27	27	24	23	507	364	59	50	31	31	2304	2090	15	15	15	4	254	229	19	17	14	14	203	193
SKIZZO	18	18	15	15	663	525	57	48	29	29	2425	2199	12	12	12	3	311	309	17	15	12	12	208	203
YQUAFFLE	28	28	25	24	490	353	59	50	31	31	2304	2090	15	15	15	4	254	229	16	14	11	11	221	215
	k						katz						s						tipdiam					
	H	H'	H'_q	$<$	μ	μ_q	H	H'	H'_q	$<$	μ	μ_q	H	H'	H'_q	$<$	μ	μ_q	H	H'	H'_q	$<$	μ	μ_q
QMRES	109	96	71	55	370	364	13	13	13	13	351	306	165	61	6	6	796	497	145	145	97	94	311	194
QUANTOR	119	106	101	53	378	372	20	20	20	18	276	240	154	50	–	–	3102	–	127	127	80	79	352	219
QUBE3.0	263	252	194	76	182	178	20	20	20	18	275	240	170	66	9	9	557	353	132	132	85	83	345	214
QUBE6.1	175	168	151	59	93	89	14	14	14	14	331	289	109	12	–	–	3471	–	52	52	20	20	546	347
SKIZZO	30	21	18	17	276	269	20	20	20	18	275	240	152	48	–	–	3223	–	70	70	46	42	395	259
YQUAFFLE	236	225	185	74	204	200	20	20	20	18	275	240	170	66	9	9	557	353	132	132	85	84	342	213

for katz encodings (about 30%) and s encodings (about 35%). Furthermore, there is no single solver dominating over all encodings: QMRES is best on add and second best on k encodings; QUANTOR is best on count (ex-aequo with SKIZZO) and third best on k encodings; QUBE6.1 is the strongest on tipdiam and s encodings – apparently due to internal preprocessing, given the performances of QUBE3.0; SKIZZO, on the other hand, is the strongest on k encodings. [7]

In the next experiment, for each solver we consider the formulas that it could not solve according to the results of Table 2 (bottom). For each solver and each such formula φ, we compute $\hat{tw}_p(\varphi)$, preprocess φ with QUBIS – setting $deg = 20$ and $div = 2000$ – to yield a new QBF φ' and then compute $\hat{tw}_p(\varphi')$ with QUTE. The goal of this experiment is to see whether QUBIS, used as a preprocessor, is able to decrease $\hat{tw}_p$ of (solver-wise) hard encodings. Table 3 shows the results. The table is split horizontally in two parts. In each part, the column "**Solver**" reports the name of a solver followed by four groups of columns, one for each encoding of Table 2. Each group contains six columns: H is the number of (solver-wise) hard formulas, H' is the number of such formulas for which QUTE was able to estimate the treewidth, H'_q is the number of such formulas preprocessed by QUBIS for which QUTE was able to estimate the treewidth, "$<$" is the number of formulas φ such that $\hat{tw}_p(\varphi') < \hat{tw}_p(\varphi)$. Columns "$\mu$" and "$\mu_q$" contain the mean value of $\hat{tw}_p$ for formulas in "H'" and "H'_q", respectively.

Looking at Table 3, we can see that, on average, preprocessing with QUBIS decreases $\hat{tw}_p$. Indeed $\mu_q < \mu$ for all solvers and all encodings in Table 3. In several cases the set of encodings for which QUBIS is able to decrease $\hat{tw}_p$ almost coincides with the set that it is able to preprocess without exceeding its resource limits. This phenomenon is most evident for the families add, circ – where the H'_q and "$<$" actually coincide for every solver – cp, and katz. In some cases, e.g., two formulas in the add group and thirteen QUANTOR-hard encodings in the k group, $\hat{tw}_p$ can be decreased of one order of magnitude. Comparatively, there are only 38 cases in which treewidth is increased. Considering the total number of literals in a formula as a size indicator, we

[7] SKIZZOis also the strongest solver overall, with 539 encodings solved, 20% more than QUBE6.1 which comes second best.

Table 4. Performances of a selection of QBF solvers on preprocessed encodings

Solver	add				circ				count				cp			
	H	*S*	#	Time	*H*	*S*	#	Time	*H*	*S*	#	Time	*H*	*S*	#	Time
QMRES	12	–	2	41.47	63	–	–	–	16	–	–	–	23	–	–	–
QUANTOR	24	–	–	–	56	–	–	–	12	–	–	–	8	–	1	41.50
QUBE3.0	27	–	1	9.04	59	–	–	–	15	–	–	–	18	–	–	–
QUBE6.1	27	–	2	8.24	59	–	–	–	15	–	15	57.32	19	–	2	412.82
SKIZZO	18	–	–	–	57	–	9	3274.30	12	–	–	–	17	–	–	–
YQUAFFLE	28	–	2	29.52	59	–	–	–	15	–	–	–	16	–	1	2.56

	k				katz				s				tipdiam			
	H	*S*	#	Time	*H*	*S*	#	Time	*H*	*S*	#	Time	*H*	*S*	#	Time
QMRES	109	19	30	1508.64	13	–	–	–	165	1	6	471.88	145	4	15	955.63
QUANTOR	119	–	35	2.76	20	–	–	–	154	–	–	–	127	3	4	67.19
QUBE3.0	263	55	88	3547.85	20	–	2	209.24	170	3	4	378.24	132	4	22	2061.39
QUBE6.1	175	14	50	4942.48	14	–	–	–	109	–	–	–	52	–	–	–
SKIZZO	30	–	1	67.54	20	–	2	172.17	152	–	–	–	70	–	10	2419.14
YQUAFFLE	236	37	63	4219.11	20	–	–	–	170	3	4	380.96	132	4	19	1048.42

found out that the average size of such formulas is about one order of magnitude smaller than the size of the ones for which QUBIS is able to decrease $\hat{tw}_p$, while the number of quantifier blocks is, on average, a factor of two higher. Indeed, the net effect of QUBIS on these kind of formulas is to increase the size of clauses without eliminating any quantifier block, which means that $\hat{tw}_p$ may increase because minimal cliques (clauses) are bigger after preprocessing and the ordering on the vertices is as constrained as it was before preprocessing.

Another relevant fact from Table 3 is that there are cases in which (i) estimating tw_p is difficult and/or (ii) preprocessing is difficult. As it turns out, for some unprocessed formulas, e.g., in the family `s`, we are not even able to compute $\hat{tw}_p$, and for other formulas, e.g., in the family `circ`, we can compute $\hat{tw}_p$ for more formulas than the ones tamed by QUBIS. Needless to say, such families are quite challenging and, on average, their members feature relatively high values of $\hat{tw}_p$. One last obervation about Table 3 is related to the fact that it may seem unlikely that a tool like QUBIS is able to consistently decrease $\hat{tw}_p$ – indeed, we have discussed cases where the converse is true. By definition of tw_p, if we let M be the size of the largest clause in a QBF φ, then we know that $M \leq tw_p(\varphi)$. Since variable elimination tends to generate large clauses, we would expect that in preprocessed formulas $\hat{tw}_p$ is higher than in the original ones. However, as discussed in [16], this is not necessarily the case, as long as the *connectivity* of φ does not increase. Clearly, for QUBIS to work properly as a preprocessor, the setting of the parameters deg and div is crucial – and not necessarily the best one in all the situations. Our setting in the above experiments reflects a good trade-off between efficiency in the usage of resources, and effectiveness in decreasing $\hat{tw}_p$.

In Table 4 we show the performances of QBF solvers on hard encodings after preprocessing. The Table is organized similarly to Table 3, and there are four columns in each group: H is the number of hard QBFs for each solver/encoding, S is the number of such formulas solved by QUBIS during preprocessing, "#" and "Time" contain, respectively, the total number of formulas solved (including the ones solved by QUBIS), and the cumulative CPU time in seconds. Looking at Table 4 (top), considering the group `add`, we can see that QMRES, QUBE6.1 and YQUAFFLE solve two previously unsolved formulas, while QUBE3.0 solves one. In particular, QMRES is able to solve

two more encodings since they had an estimated quantified treewidth of 658 and 943 but QuBIS decreased it to 77 and 93, respectively. Considering the group `circ`, we can see that only sKizzo is able to solve formulas (9 out of 29) that it found hard before preprocessing. Considering the group `count`, as in the case of `circ`, we see that only QuBE6.1 takes advantage of preprocessing by solving 15 previously unsolved formulas. Looking now at the `cp` group, we see that QuBE6.1 solves 2 hard formulas, while both quantor and yQuaffle solve 1 hard formula.

Still with reference to Table 4, we consider now the encodings on the bottom. In the group `k`, preprocessing turns out to be very effective for most solvers, with the exception of sKizzo because of a ceiling effect: indeed, sKizzo alone is already quite effective on such formulas. On the other hand, search-based solvers benefit the most, and it is fair to say that QuBIS complements the shortcomings of these solvers on such encodings. However, it is interesting to notice that also variable-elimination based solvers like quantor and QMRes improve their performances, which contributes to the thesis that decreasing $\hat{tw}_p$ is useful independently from the specific algorithm featured by the solver. As for the group `katz`, only QuBE3.0 and sKizzo are able to solve 2 previously unsolved formulas. Similar results hold also for the `s` group, wherein QMRes is the solver which benefits the most from preprocessing. The group `tipdiam` is quite interesting in its own, since most solvers are able to benefit from preprocessing, again in an algorithm independent fashion.

We conclude the analysis of Table 4 by mentioning that, overall, the dataset therein considered consists of 272 hard – in the sense of Section 3 – formulas. QuBIS can preprocess 113 such formulas in a successful way, i.e., without exceeding its resource bounds. Noticeably, considering an ideal solver that always fares the best result for each pair solver/encoding *after* preprocessing, we have that 25 previously hard formulas can now be solved. Considering that QuBIS is still a proof-of-concept implementation, we view this as an indication that preprocessing geared towards reducing quantified treewidth is an enabler to deal with challenging QBF encodings.

5 Related Work and Conclusions

The empirical role of treewidth has been previously explored in the CSP [26] and SAT [16] literature. Before this paper, there was no such study in QBF, albeit the papers about quantor and QMRes (see,e.g., [15,14]) implicitly leverage the same concepts and are thus related to our contribution. From a theoretical standpoint, there are two other papers [5,6] that together with [4] consider the relationship between structural restrictions of QBFs and the complexity of reasoning about them. Here, we follow [4] because its characterization of tw_p accounts nicely for the structure of the prefix and the structure of the formula in a single parameter. In [6], the prefix is taken into account by considering alternation depth, while treewidth accounts only for the structure of the matrix. In [5] a completely different kind of structural restriction, which is also incomparable with treewidth, is presented. From an experimental point of view, it may be interesting to check how the results of [6] and [5] apply to our setting, and whether they shed further insight on the behaviour of QBF solvers on hard encodings.

Concerning QuBIS, our direct source of inspiration has been the algorithm of Bounded Directional Resolution (BDR) presented in [16]. From an implementation point of view, even as a proof-of-concept implementation, QuBIS is more advanced than BDR. From an algorithmic point of view, the main difference between BDR and QuBIS is that our solver uses dynamic, rather than static, reordering of variables. Since QuBIS uses Q-resolution to eliminate variables, it is in this aspect similar to QMRes [15] and QUANTOR [14]. However, our approach differs from QMRes, because we do not use symbolic data structures, and also from QUANTOR since we never expand universal variables.

In this paper we have studied the practical relevance of $\hat{tw}_p$ as a marker of empirical hardness in QBFs. We have shown that such approximation is, in a statistical sense, a robust predictor of the difficulty encountered by solvers facing QBF encodings. We have shown that other purely syntactic features, alone or in combination among them, are not as good as $\hat{tw}_p$. Finally, we have shown that decreasing $\hat{tw}_p$ as done by QuBIS can enable QBF solvers to cope with hard encodings. Our future work will include looking for more accurate bounds when approximating tw_p, evaluating the impact of other preprocessors for QBFs on tw_p, and an evaluation of tw_p as a control parameter for multi-engine solvers.

References

1. Bodlaender, H.L.: Treewidth: Characterizations, applications, and computations. Technical report, Utrecht University (2006)
2. Freuder, E.: Complexity of k-tree structured constraint satisfation problem. In: Proc. of AAAI 1990 (1990)
3. Dechter, R., Pearl, J.: Tree Clustering for constraint networks. Artificial Intelligence, 61–95 (1989)
4. Chen, H., Dalmau, V.: From pebble games to tractability: An ambidextrous consistency algorithm for quantified constraint satisfaction. In: Ong, L. (ed.) CSL 2005. LNCS, vol. 3634. Springer, Heidelberg (2005)
5. Gottlob, G., Greco, G., Scarcello, F.: The Complexity of Quantified Constraint Satisfaction Problems under Structural Restrictions. In: IJCAI 2005, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, pp. 150–155. Professional Book Center (2005)
6. Pan, G., Vardi, M.Y.: Fixed-Parameter Hierarchies inside PSPACE. In: 21th IEEE Symposium on Logic in Computer Science (LICS 2006), pp. 27–36. IEEE Computer Society, Los Alamitos (2006)
7. Stockmeyer, L.J., Meyer, A.R.: Word problems requiring exponential time. In: 5th Annual ACM Symposium on the Theory of Computation, pp. 1–9 (1973)
8. Giunchiglia, E., Narizzano, M., Pulina, L., Tacchella, A.: Quantified Boolean Formulas satisfiability library, QBFLIB (2001), www.qbflib.org
9. Narizzano, M., Pulina, L., Taccchella, A.: QBF solvers competitive evaluation (QBFEVAL) (2006), http://www.qbflib.org/qbfeval
10. Arnborg, S., Corneil, D.G., Proskurowski, A.: Complexity of finding embeddings in a k-tree. SIAM Journal on Algebraic and Discrete Methods, 277–284 (1987)
11. Giunchiglia, E., Narizzano, M., Tacchella, A.: Clause-Term Resolution and Learning in Quantified Boolean Logic Satisfiability. Artificial Intelligence Research 26, 371–416 (2006), http://www.jair.org/vol/vol26.html

12. Zhang, L., Malik, S.: Conflict driven learning in a quantified boolean satisfiability solver. In: Proceedings of International Conference on Computer Aided Design (ICCAD 2002) (2002)
13. Benedetti, M.: sKizzo: A suite to evaluate and certify QBFS. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS, vol. 3632, pp. 369–376. Springer, Heidelberg (2005)
14. Biere, A.: Resolve and expand. In: Hoos, H., Mitchell, D.G. (eds.) SAT 2004. LNCS, vol. 3542, pp. 59–70. Springer, Heidelberg (2005)
15. Pan, G., Vardi, M.Y.: Symbolic decision procedures for QBF. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 453–467. Springer, Heidelberg (2004)
16. Rish, I., Dechter, R.: Resolution versus search: Two strategies for sat. Journal of Automated Reasoning 24(1/2), 225–275 (2000)
17. Tarjan, R.E., Yannakakis, M.: Addendum: Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. SIAM J. Comput. 14(1), 254–255 (1985)
18. Gogate, V., Dechter, R.: A Complete Anytime Algorithm for Treewidth. In: UAI 2004, Proceedings of the 20th Conference in Uncertainty in Artificial Intelligence, pp. 201–208. AUAI Press (2004)
19. Subbarayan, S., Andersen, H.R.: Backtracking Procedures for Hypertree, HyperSpread and Connected Hypertree Decomposition of CSPs. In: IJCAI, pp. 180–185 (2007)
20. Benedetti, M.: Quantifier trees for QBFS. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569. Springer, Heidelberg (2005)
21. Giunchiglia, E., Narizzano, M., Tacchella, A.: Quantifier Structure in search based procedures for QBFs. IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems 26(3) (2007)
22. Pulina, L., Taccchella, A.: MIND-Lab projects and related information (2008), http://www.mind-lab.it/projects
23. Pulina, L., Tacchella, A.: A multi-engine solver for quantified boolean formulas. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 574–589. Springer, Heidelberg (2007)
24. Witten, I.H., Frank, E.: Data Mining, 2nd edn. Morgan Kaufmann, San Francisco (2005)
25. Kleine-Büning, H., Karpinski, M., Flögel, A.: Resolution for Quantified Boolean Formulas. Information and Computation 117(1), 12–18 (1995)
26. Larrosa, J., Dechter, R.: Boosting Search with Variable Elimination in Constraint Optimization and Constraint Satisfaction Problems. Constraints 8(3), 303–326 (2003)

Tractable Quantified Constraint Satisfaction Problems over Positive Temporal Templates*

Witold Charatonik and Michał Wrona

Institute of Computer Science
University of Wrocław

Abstract. A positive temporal template (or a positive temporal constraint language) is a relational structure whose relations can be defined over a dense linear order of rational numbers using a relational symbol $\leq$, logical conjunction and disjunction.

We provide a complexity characterization for quantified constraint satisfaction problems (*QCSP*) over positive temporal languages. The considered *QCSP* problems are decidable in LOGSPACE or complete for one of the following classes: NLOGSPACE, P, NP, PSPACE. Our classification is based on so-called algebraic approach to constraint satisfaction problems: we first classify positive temporal languages depending on their surjective polymorphisms and then give the complexity of *QCSP* for each obtained class.

The complete characterization is quite complex and does not fit into one paper. Here we prove that *QCSP* for positive temporal languages is either NP-hard or belongs to P and we give the whole description of the latter case, that is, we show for which positive temporal languages the problem *QCSP* is in LOGSPACE, and for which it is NLOGSPACE-complete or P-complete. The classification of NP-hard cases is given in a separate paper.

1 Introduction

Constraint Satisfaction Problems provide a uniform approach to research on a wide variety of combinatorial problems. Besides probably better-known *CSP* over finite domains [9,11,19] with its Dichotomy conjecture of Feder and Vardi [12], *CSP* over infinite domains are of more and more interest. Although there were some earlier results in this field [1,16], a common approach to *CSP* over infinite domains was quite recently proposed and developed by Manuel Bodirsky [2] and co-authors. This framework concentrates on relational structures that are ω-categorical [14]. Many results, including so-called algebraic approach [15,11], for both *CSP* and *QCSP* [8] over finite domains were generalized to infinite ones. Moreover, new results were established. Among them there are full characterizations of complexity for both *CSP* and *QCSP* of equality constraint languages [6,4].

As each natural theoretical framework *CSP* have many different applications. It is also the case in the area of *CSP* with infinite templates. For example, in [3,5] it is argued that *CSP* of relations definable over the dense linear order of rational numbers

* Work partially supported by Polish Ministry of Science and Education grant 3 T11C 042 30.

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 543–557, 2008.

may constitute a good approach to temporal and spatial reasoning [13,18]. Therefore the very recent full complexity characterization of *CSP* for temporal, that is, definable over $\langle \mathbb{Q}, < \rangle$ templates gives a new perspective on temporal reasoning [7]. One should mention in this context also the well-motivated AND/OR precedence constraints [17]. They are closely related to languages from items 3 and 4 of Theorem 1 below. It might be said that we consider quantified positive variations of AND/OR precedence constraints.

Our paper is the next step in the area of quantified constraint satisfaction problems over temporal templates. In general, we consider *QCSP* for temporal templates. In particular, we restrict ourselves to constraint languages that can be defined with $\wedge, \vee$ and $\leq$, i.e., we do not consider negation. We name such relations positive temporal since they are positive definable over $\langle \mathbb{Q}, \leq \rangle$.

Our main contribution is a complexity characterization of QCSP problems over positive temporal languages summarized in Theorem 1 below.

Theorem 1 (The Main Theorem). *Let Γ be a language of positive temporal relations, then one of the following holds.*

1. *Each relation in Γ is definable by a conjunction of equations ($x_1 = x_2$) and then QCSP(Γ) is decidable in LOGSPACE.*
2. *Each relation in Γ is definable by a conjunction of weak inequalities ($x_1 \leq x_2$). If there exists a relation in Γ that is not definable as a conjunction of equalities, then QCSP(Γ) is NLOGSPACE-complete.*
3. *Each relation in Γ is definable by a formula of the form $\bigwedge_{i=1}^{n}(x_{i_1} \leq x_{i_2} \vee \ldots \vee x_{i_1} \leq x_{i_k})$ and then provided Γ satisfies neither condition 1 nor 2, the set QCSP(Γ) is P-complete.*
4. *Each relation in Γ is definable by a formula of the form $\bigwedge_{i=1}^{n}(x_{i_2} \leq x_{i_1} \vee \ldots \vee x_{i_k} \leq x_{i_1})$ and then provided Γ satisfies neither condition 1 nor 2, the set QCSP(Γ) is P-complete.*
5. *Each relation in Γ is definable by a formula of the form $\bigwedge_{i=1}^{n}(x_{i_1} = y_{i_1} \vee \ldots \vee x_{i_k} = y_{i_k})$ and then provided Γ does not belong to any of the classes 1–4, the set QCSP(Γ) is NP-complete.*
6. *The problem QCSP(Γ) is PSPACE-complete.*

The complete characterization is quite complex and does not fit into one paper. Therefore some parts of Theorem 1 are proved in a companion paper [10]. Technically, in this paper we show the following.

Theorem 2. *If Γ is a positive temporal language, then either it satisfies one of conditions 1–4 of Theorem 1 or QCSP(Γ) is NP-hard.*

Related work. As we already mentioned, parts of Theorem 1 are proved in a different paper [10]. We give there the characterization of NP-hard cases, with the distinction between NP-complete and PSPACE-complete cases (items 5 and 6 of Theorem 1) and the complexity proofs for items 3 and 4 of Theorem 1 (the algebraic characterization of these cases is given here in Section 5). In [10] we use different techniques, in particular

we do not use there the filter representation of temporal relations which is our basic tool in this paper.

The characterizations of cases 1, 2 and 5 are due to other authors [4,3]. Here, we just complete the picture by giving the NLOGSPACE-hardness proof for case 2. Theorem 1 substantially improves results from [4,3] in the sense that we consider a strictly more expressive class of constraint languages. As in [4] we use the surjective preservation theorem.

In a very recent paper [7] the authors give a classification of *CSP* over temporal languages depending on their polymorphisms. Although it sounds similar, it is different from our classification. We deal with positive temporal languages and surjective polymorphisms, which are used to classify *QCSP* problems (as opposed to *CSP* problems considered in [7]). In the case of positive temporal languages, the classification based on polymorphisms is trivial: all these languages fall into the same class because they are all closed under constant functions — as a consequence all *CSP* problems for positive temporal languages are trivial. To obtain our classification we use methods different from those used in [7]. The representation of temporal relations from Section 3 and the proof in sections 5 and 6 plays a similar role in the proof of Theorem 1 as the Ramsey Theorem in the proof from [7] in the sense that it is used to show that there are only few interesting classes of positive temporal relations closed under non-unary functions. Nevertheless our representation may be easily translated into first order formula. It is very useful in the context of considered *QCSP* problems — complexity proofs for families of positive temporal templates from Theorem 1 are purely syntactical [4,3,10].

Outline of the paper. In Section 2, we give some preliminaries. Among others, we recall a definition of a surjective polymorphism and surjective preservation theorem, which is the most important tool in algebraic approach to *QCSP*. In Section 3 we propose a representation of temporal relations. Section 4 is devoted to formally introducing the concept of positive temporal relations. By giving their properties we unfold the reason we choose just that subset. In Section 5, we derive first four items of Theorem 1, in Section 6 we prove that all other languages are NP-hard and finally the last section is devoted to the proof of Theorem 2.

2 Preliminaries

In most cases we follow the notation from [2,4].

Relational structures. We consider only relations defined over countable domains and hence whenever we write a domain or D we mean a countable set. Let τ be some relational (in this paper always finite) signature i.e., a set of relational symbols with assigned arity. Then Γ is a τ-structure over domain D if for each relational symbol R_i from τ, it contains a relation of according arity defined on D. In the rest of the paper we usually say relational language (or template) instead of relational structure. Moreover, we use the same notation for relational symbols and relations.

Automorphisms of Γ constitute a group with respect to composition. An orbit of a k-tuple t in Γ is the set of all tuples of the form $\langle \Pi(t_1), \ldots, \Pi(t_k) \rangle$ for all automorphisms Π. We say that a group of automorphisms of Γ is oligomorphic if for each k it

has a finite number of orbits of k-tuples. Although there are many different ways of introducing a concept of ω-categorical structures we do it by the following theorem [14].

Theorem 3. ***(Engeler, Ryll-Nardzewski, Svenonius)*** *Let Γ be a relational structure. Then Γ is ω-categorical if and only if the automorphism group of Γ is oligomorphic.*

Polymorphisms and clones. Let R be a relation of arity n defined over D. We say that a function $f : D^m \to D$ is a polymorphism of R if for all $a^1, \ldots, a^m \in R$ (where a^i, for $1 \leq i \leq m$, is a tuple $\langle a^i_1, \ldots, a^i_n \rangle$), we have $\langle f(a^1_1, \ldots, a^m_1), \ldots, f(a^1_n, \ldots, a^m_n) \rangle \in R$. Then we say that f preserves R or that R is closed under f. A polymorphism of Γ is a function that preserves all relations of Γ. By *Pol*(Γ) we denote the set of polymorphisms of Γ, and by *sPol*(Γ) — the set of surjective polymorphisms.

An operation π is a projection iff $\pi(x_1, \ldots, x_m) = x_i$ for all m-tuples and fixed $i \in \{1, \ldots, m\}$. The set of polymorphisms of an ω-categorical language Γ constitute a *clone*, that is, a set of functions closed under composition and containing all projections. We say that a function f with a domain D is an *interpolation* of a set of functions F iff for every finite subset B of D there is some operation $g \in F$ such that $f(a) = g(a)$ for every tuple a over the set B. The set of interpolations of F is called the *local closure* of F. We say that a clone is *locally closed* if each its subset contains its local closure. For each ω-categorical language Γ the clone of its polymorphisms is locally closed [2]. A clone is generated by a set of functions F if it is the least clone containing F.

An operation f of arity m is *essentially unary* if there exists a unary operation f_0 such that $f(x_1 \ldots, x_m) = f_0(x_i)$ for some fixed $i \in \{1, \ldots, m\}$. An operation that is not essentially unary is called *essential*. We say that a polymorphism f of an ω-categorical structure Γ is *oligopotent* if the diagonal of f, that is, the function $f(x, \ldots, x)$, is contained in the locally closed clone generated by the automorphisms of Γ. A function f is called a *quasi near-unanimity function* (QNUF) if it satisfies $f(x, \ldots, x, y) = f(x, \ldots, y, x) = \ldots = f(y, x, \ldots, x) = f(x, \ldots, x)$ for all $x, y \in D$.

Quantified constraint satisfaction problems. Let Γ contain $R_1, \ldots, R_k$. Then a conjunctive positive formula (*cp-formula*) over Γ is a formula of the following form:

$$Q_1 x_1 \ldots Q_n x_n (R_1(\boldsymbol{v_1}) \wedge \ldots \wedge R_k(\boldsymbol{v_k})), \tag{1}$$

where $Q_i \in \{\forall, \exists\}$ and $\boldsymbol{v_j}$ are vectors of variables $x_1, \ldots, x_n$.

A *QCSP*(Γ) is a problem to decide whether a given cp-formula without free variables over Γ is true or not. Note that by downward Löwenheim-Skolem Theorem we can focus on countable domains only. If all quantifiers in (1) are existential then the corresponding problem is well-known as the *constraint satisfaction problem*.

A relation R has a *cp-definition* in Γ if there exists a cp-formula $\phi(x_1, \ldots, x_n)$ over Γ such that for all $a_1, \ldots, a_n$ we have $R(a_1, \ldots, a_n)$ iff $\phi(a_1, \ldots, a_n)$ is true. The set of all relations cp-definable in Γ is denoted by $[\Gamma]$.

Lemma 1 ([4]). *Let Γ_1, Γ_2 be relational languages. If every relation in Γ_1 has a cp-definition in Γ_2, then QCSP(Γ_1) is log-space reducible to QCSP(Γ_2).*

The following results link $[\Gamma]$ with *sPol*(Γ). The idea behind Theorem 4 is that the more Γ can express, in the sense of cp-definability, the less polymorphisms are

contained in $sPol(\Gamma)$. Moreover, the converse is also true. This theorem is called *surjective preservation theorem*.

Theorem 4 ([4]). *Let Γ be an ω-categorical structure. Then a relation R has a cp-definition in Γ if and only if R is preserved by all surjective polymorphisms of Γ.*

As a direct consequence of Lemma 1 and Theorem 4 we obtain the following.

Corollary 1 ([4]). *Let Γ_1, Γ_2 be ω-categorical structures. If $sPol(\Gamma_2) \subseteq sPol(\Gamma_1)$, then $QCSP(\Gamma_1)$ is log-space reducible to $QCSP(\Gamma_2)$.*

Quantified Equality Constraints. Concerning templates that allow equalities and all logical connectives (equality constraint languages) the following classification [4] is known.

1. **Negative languages**. Relations of such a language are definable as CNF-formulas whose clauses are either equalities ($x = y$) or disjunctions of disequalities ($x_1 \neq y_1 \vee \ldots \vee x_k \neq y_k$). For each negative Γ the problem $QCSP(\Gamma)$ is contained in LOGSPACE.
2. **Positive languages**. Relations may be defined as conjunction of disjunctions of equalities ($x_1 = y_1 \vee \ldots \vee x_k = y_k$). For each positive Γ not being negative the problem $QCSP(\Gamma)$ is NP-complete.
3. In any other case the problem $QCSP(\Gamma)$ is PSPACE-complete.

Note that the class 1 from Theorem 1 is a subset of negative languages and the class 5 is just the class of positive languages.

To give our characterization we need the following result. It may be inferred from lemmas given in Section 7 in [4].

Lemma 2. *Let Γ be an equality positive constraint language that is preserved by an essential operation on D with infinite image. Then Γ is preserved by all operations, and Γ is negative.*

Corollary 2. *If an equality constraint language Γ is positive, but not negative, then $sPol(\Gamma)$ contains only essentially unary polymorphisms.*

Since $QCSP(\Gamma)$ for positive non-negative equality language Γ is NP-hard, by corollaries 1 and 2, we obtain one more observation.

Corollary 3. *Let Γ be a positive temporal language with $sPol(\Gamma)$ contained in the set of essentially unary surjections on $\mathbb{Q}$. Then $QCSP(\Gamma)$ is NP-hard.*

Quantified Positive Temporal Constraints. Now, we focus on positive temporal relations announced in the introduction. All of them are defined over the set of rational numbers using a relational symbol $\leq$ and connectives $\wedge, \vee$. Therefore our results concerning positive temporal relations generalize those for positive equality languages. Since the only relational symbol we use is interpreted as a weak linear order over rational numbers, for each positive temporal structure Γ the set $sPol(\Gamma)$ contains all automorphisms

that preserve order, i.e., all increasing unary surjections $f : \mathbb{Q} \to \mathbb{Q}$. We say that f is increasing (weakly increasing) if $f(a) > f(b)$ ($f(a) \geq f(b)$) for all $a > b$. Thus, using Theorem 3, it is not hard to see that all positive temporal languages are ω-categorical.

In Lemma 6 in Section 4 we provide another, not syntactical, characterization of positive temporal relations. It is given in terms of polymorphisms.

3 Filter Representation of Temporal Relations

Here, after a few definitions we give a representation of temporal languages that we use in the rest of the paper. At first look it may look somewhat confusing, but Example 1 and a short discussion after it should clarify our point.

A preorder is a reflexive and transitive relation. A preorder $\preceq$ on a set A is total if for all $a, b \in A$ we have $a \preceq b$ or $b \preceq a$. We call A the domain of $\preceq$ and we write $A = Dom(\preceq)$. We use $a \prec b$ as an abbreviation for $a \preceq b \wedge b \not\preceq a$ and $a \approx b$ as an abbreviation for $a \preceq b \wedge b \preceq a$. In the following we represent total preorders on finite sets of variables as sequences of the form $x_1 \sim_1 x_2 \sim_2 \ldots \sim_{n-1} x_n$ where each $\sim_i$ is either $\prec$ or $\approx$ and $\{x_1, \ldots, x_n\} = Dom(\preceq)$. For example $a \prec b \approx c$ is the representation of $\preceq = \{\langle a, a\rangle, \langle b, b\rangle, \langle c, c\rangle, \langle a, b\rangle, \langle a, c\rangle, \langle b, c\rangle, \langle c, b\rangle\}$. By *Range*$(\preceq)$ we denote the size of a maximal set of variables $\{x_{i_1}, \ldots, x_{i_l}\} \subseteq Dom(\preceq)$ such that for all pairs x_{i_p}, x_{i_r} we have either $x_{i_p} \prec x_{i_r}$ or $x_{i_r} \prec x_{i_p}$.

We write $\preceq_1 \triangleleft \preceq_2$ and say that $\preceq_1$ is more general than $\preceq_2$ if $\preceq_1$ is a restriction of $\preceq_2$ to a smaller domain. Formally, $\preceq_1 \triangleleft \preceq_2$ if $Dom(\preceq_1) \subseteq Dom(\preceq_2)$ and $\preceq_1 = \preceq_2 \cap (Dom(\preceq_1) \times Dom(\preceq_1))$. We write $\preceq_1 \ll \preceq_2$ and say that $\preceq_1$ is flatter than $\preceq_2$ if $Dom(\preceq_1) = Dom(\preceq_2)$ and $\preceq_2$ as a relation is a subset of $\preceq_1$ (see the example below).

We say that a valuation $q : \{x_1, \ldots, x_n\} \to \mathbb{Q}$ is compatible with a total preorder $\preceq$ on $\{x_1, \ldots, x_n\}$ if for all x_i, x_j such that $x_i \preceq x_j$ we have $q(x_i) \leq q(x_j)$. We then also say that the tuple $\langle q(x_1), \ldots, q(x_n)\rangle$ is compatible with $\preceq$.

Definition 1. *Consider a temporal relation $R(x_1, \ldots, x_n) \subseteq \mathbb{Q}^n$. We say that $\preceq$ is a* bound *for R if $\preceq$ is a total preorder on a subset of $\{x_1, \ldots, x_n\}$ such that for all valuations $q : \{x_1, \ldots, x_n\} \to \mathbb{Q}$ compatible with $\preceq$ the tuple $\langle q(x_1), \ldots, q(x_n)\rangle$ is* not *in R. The set of bounds of R is denoted $\mathcal{B}(R)$. A minimal wrt. $\triangleleft$ bound of R is called a* filter *for R. The set of filters of R is denoted $\mathcal{F}(R)$.*

Let Γ be a temporal template. Assume that each $R \in \Gamma$ is defined over different set of variables. Then by $\mathcal{F}(\Gamma)$ equal to $\bigcup_{R \in \Gamma} \mathcal{F}(R)$ we denote the set of filters of Γ. Similarly, we write $\mathcal{B}(\Gamma)$ for the set of bounds of Γ.

Example 1. Let $R(x_1, x_2, x_3)$ be a relation given by $(x_1 \leq x_2 \vee x_2 \leq x_3) \wedge (x_2 \leq x_1)$. Then $x_3 \prec x_2 \prec x_1$, $x_1 \prec x_2$ and $x_1 \approx x_3 \prec x_2$ are bounds for R (in fact R has more bounds). The bound $x_1 \approx x_3 \prec x_2$ is not a filter because $x_1 \prec x_2$ is more general. The relation R' defined by $(x_1 \leq x_2 \vee x_1 \leq x_3)$ has three filters: $x_3 \prec x_2 \prec x_1$, $x_3 \approx x_2 \prec x_1$ and $x_2 \prec x_3 \prec x_1$. The filter $x_3 \approx x_2 \prec x_1$ is flatter than both $x_3 \prec x_2 \prec x_1$ and $x_2 \prec x_3 \prec x_1$. The range of a preorder $x_3 \approx x_2 \prec x_1$ is 2, but the range of a preorder $x_3 \prec x_2 \prec x_1$ is equal to 3.

In the following we represent temporal relations (languages) with their sets of filters. It is quite simple to infer $\mathcal{F}(R)$ from the cp-definitions in Example 1. Therefore one would ask why we do not represent relations with their sets of clauses. The filter representation gives us a kind of normal form while there may be many representations of the same relations with sets of clauses. The following example shows another advantage of filters. The relation R defined by $(x_1 \geq x_2 \vee x_2 \geq x_3)$ has one filter: $x_1 \prec x_2 \prec x_3$. The clause representation $(x_1 \geq x_2 \vee x_2 \geq x_3)$ does not say anything about dependencies between x_1 and x_3. From the shape of the filter it is easy to see that R contains tuples compatible with preorders: $x_1 \prec x_2, x_1 \prec x_3, x_2 \prec x_3$ but it does not contain any tuple compatible with $x_1 \prec x_2 \prec x_3$. We use this property in several proofs.

Consider now the situation where we have some $\mathcal{F}(\Gamma)$ and we ask for $\mathcal{F}(R)$ of some relation R that is cp-definable over Γ; and the converse situation: when we want to infer something about $\mathcal{F}(\Gamma)$ from $\mathcal{F}(R)$. The following lemmas give us a partial answer for such questions. In lemmas 3–5 the relation R_1 belongs to $[R]$. These lemmas are used in each of the following sections of the paper and therefore are of crucial importance.

Let $\preceq_{\mathit{Var}}$ be a preorder with domain *Var* such that $x \approx y$ for all $x, y \in \mathit{Var}$.

Lemma 3. *Let $R(x_1, \ldots, x_n)$ be a temporal relation. Consider $R_1(x_k, \ldots, x_n)$ defined by $R(x_1, \ldots, x_n) \wedge \bigwedge_{i=1}^{k-1} x_i = x_{i+1}$. If a preorder $\preceq_1$ is a bound of R_1, then each preorder $\preceq$ such that $\preceq_1 \lhd \preceq$ and $x_i \approx x_{i+1}$ for $1 \leq i \leq k-1$ is a bound of R.*

Lemma 4. *Let $R(x_1, \ldots, x_n)$ be a temporal relation with a filter $\preceq$ such that $x_i \approx x_{i+1}$ for $i = 1, \ldots, k-1$. Consider $R_1(x_k, \ldots, x_n)$ defined by $R(x_1, \ldots, x_n) \wedge \bigwedge_{i=1}^{k-1} x_i = x_{i+1}$. Then either the restriction of $\preceq$ to $\{x_k, \ldots, x_n\}$ or the restriction of $\preceq$ to $\{x_{k+1}, \ldots, x_n\}$ is a filter of R_1.*

Example 2. Consider the preceding lemma once more. At first glance it may seem that a restriction of $\preceq$ to $\{x_k, \ldots, x_n\}$ is always a filter of R_1 and the second case is unnecessary there. To see that it is not the case, consider the following relation.

Let $R(x_1, x_2, x_3, x_4)$ be defined by $(x_1 = x_2 \vee x_1 = x_3 \vee x_1 = x_4) \wedge (x_2 = x_1 \vee x_2 = x_3 \vee x_2 = x_4) \wedge (x_3 = x_1 \vee x_3 = x_2 \vee x_3 = x_4) \wedge (x_4 = x_1 \vee x_4 = x_2 \vee x_4 = x_3)$. Then $x_1 \approx x_2 \approx x_3 \prec x_4$ is a filter of R. Moreover, let a relation $R_1(x_2, x_3, x_4)$ be equivalent to $R(x_1, x_2, x_3, x_4) \wedge x_1 = x_2$. Now, the reader can convince himself that $x_3 \prec x_4$ and not $x_2 \approx x_3 \prec x_4$ is a filter of R_1.

Lemma 5. *Consider $R(x_1, \ldots, x_n)$ and $R_1(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$ defined by $\exists x_i\ R(x_1, \ldots, x_n)$. Then a preorder $\preceq$ such that $x_i \notin \mathit{Dom}(\preceq)$ is a filter of R if and only if it is a filter of R_1.*

4 Positive Temporal Relations

An arbitrary relation defined over the dense linear order of rational numbers is closed under all unary strictly increasing functions — they are automorphisms of such a relation. In the case of positive temporal relations we have a bit stronger result.

Lemma 6. *A temporal relation $R(x_1, \ldots, x_n)$ is positive if and only if it is closed under all weakly increasing functions $f : \mathbb{Q} \to \mathbb{Q}$.*

It is rather obvious that positive temporal relations have substantially less varied structure than arbitrary temporal relations. The dependencies between tuples in positive temporal relations can be given in terms of the order $\ll$. The following results unfold the reason why we introduced this order. Recall: if $\preceq \ll \preceq_1$, then we say that $\preceq$ is flatter than $\preceq_1$.

Lemma 7. *Let a preorder $\preceq$ be flatter than $\preceq_1$ and $Dom(\preceq) \subseteq \{x_1, \ldots, x_n\}$. Consider a tuple $\langle q(x_1), \ldots, q(x_n)\rangle$ compatible with $\preceq_1$ where $q : \{x_1, \ldots, x_n\} \to \mathbb{Q}$. Then there exists a weakly increasing function f such that $\langle f(q(x_1)), \ldots, f(q(x_n))\rangle$ is compatible with $\preceq$.*

Lemma 8. *Let $R(x_1, \ldots, x_n)$ be a temporal relation. Then it is a positive temporal relation if and only if for all bounds $\preceq$ of R each preorder $\preceq_1$ such that $\preceq \ll \preceq_1$ is also a bound of R.*

Corollary 4. *Let $R(x_1, \ldots, x_n)$ be a positive temporal relation and let the set Var be a nonempty subset of $\{x_1, \ldots, x_n\}$. Then the preorder $\preceq_{Var}$ is a bound of R if and only if R is the empty relation. Intuitively, nonempty relations do not have filters of the form $\preceq_{Var}$.*

Note that if we have two bounds comparable wrt $\ll$ then by Lemma 8 the flatter of them is more interesting. Therefore we focus on minimal wrt $\ll$ filters. We illustrate this lemma on the following example.

Example 3. Consider once more a positive temporal relation given by $(x_1 \leq x_2 \vee x_1 \leq x_3)$. It has a filter $x_2 \approx x_3 \prec x_1$, but also filters (bounds) $x_2 \prec x_3 \prec x_1$ and $x_3 \prec x_2 \prec x_1$. Note or recall from Example 1 that the first of the mentioned preorders is flatter than the remaining two. Moreover, these are all filters of the relation $(x_1 \leq x_2 \vee x_1 \leq x_3)$.

We finish this section by one more auxiliary lemma.

Lemma 9. *Let $R(x_1, \ldots, x_n)$ be a temporal relation. Let $\preceq$ be a minimal with respect to $\ll$ filter of R such that $x_i \approx x_{i+1}$ for $i = 1, \ldots, k-1$. Let $R_1(x_k, \ldots, x_n)$ be defined as $R(x_1, \ldots, x_n) \wedge \bigwedge_{i=1}^{k-1} x_i = x_{i+1}$. Then there exists a preorder $\preceq_1$ in $\mathcal{F}(R_1)$ such that $\preceq_1 \lhd \preceq$ and $Range(\preceq_1) = Range(\preceq)$.*

5 Non Unary Surjective Polymorphisms of Positive Temporal Relations

In this section we derive the first four classes of Theorem 1. We show that a positive temporal language belongs to one of these four classes, or it is closed under essentially unary surjective polymorphisms only and hence, by Corollary 3, in that case the problem $QCSP(\Gamma)$ is NP-hard.

In particular we show that a positive temporal language is closed under a binary surjective polymorphism *spp*, under a binary surjective polymorphism *dual-spp*, or it is preserved by essentially unary surjective polymorphisms only. The surjective polymorphisms *spp* : $\mathbb{Q} \times \mathbb{Q} \to \mathbb{Q}$ and *dual-spp* : $\mathbb{Q} \times \mathbb{Q} \to \mathbb{Q}$ are surjective counterparts of *pp* and *dual-pp* operations introduced in [7].

Recall that all countable dense linear orders without endpoints are isomorphic. In particular, $\mathbb{Q}, \mathbb{Q}_-, \mathbb{Q}_+$ and $\mathbb{Q}_- \cup \mathbb{Q}_+$ are isomorphic, where $\mathbb{Q}_- = \{q \in \mathbb{Q} \mid q < 0\}$ and $\mathbb{Q}_+ = \{q \in \mathbb{Q} \mid q > 0\}$. Let $f_1 : \mathbb{Q} \to \mathbb{Q}_-$, $f_2 : \mathbb{Q} \to \mathbb{Q}_+$ and $f : \mathbb{Q}_- \cup \mathbb{Q}_+ \to \mathbb{Q}$ be any order-preserving bijections. Let

$$spp'(a,b) = \begin{cases} a & \text{if } a < 0 \\ f_2(b) & \text{if } a \geq 0 \end{cases} \qquad \textit{dual-spp}'(a,b) = \begin{cases} f_1(b) & \text{if } a \leq 0 \\ a & \text{if } a > 0 \end{cases}$$

and define $spp(a,b) = f(spp'(a,b))$ and $\textit{dual-spp}(a,b) = f(\textit{dual-spp}'(a,b))$. Observe that if $spp(a,b)$ is a (strict) lower bound of $\{spp(a_1,b_1),\dots,spp(a_k,b_k)\}$ then either a is a (strict) lower bound of $\{a_1,\dots,a_k\}$ or b is a (strict) lower bound of $\{b_1,\dots,b_k\}$. Similarly, if $\textit{dual-spp}(a,b)$ is a (strict) upper bound of the set $\{\textit{dual-spp}(a_1,b_1),\dots,\textit{dual-spp}(a_k,b_k)\}$ then either a is a (strict) upper bound of $\{a_1,\dots,a_k\}$ or b is a (strict) upper bound of $\{b_1,\dots,b_k\}$.

For each $i \geq 2$ let R^i_{Left} be the positive temporal relation defined by the formula $(x_1 \leq x_2 \vee \ldots \vee x_1 \leq x_i)$. Let Γ_{Left} be the positive temporal language containing R^i_{Left} for each i. Similarly, Γ_{Right} is the set of relations defined by formulas $(x_2 \leq x_1 \vee \ldots \vee x_i \leq x_1)$. Each such formula is denoted by R^i_{Right}.

5.1 Classes of Different Power of Cp-Definability

The topic of this subsection is summarized by the following theorem. Recall from Section 2 that $[\Gamma]$ is the set of all relations that are cp-definable by relations of Γ. Note that by Lemma 1, the problems $QCSP(\Gamma_1)$ and $QCSP(\Gamma_2)$ for some positive temporal languages Γ_1 and Γ_2 such that $[\Gamma_1] = [\Gamma_2]$ are logspace equivalent.

Theorem 5. *Let Γ be a positive temporal language, then exactly one of the following holds.*

1. *Each relation in Γ is definable by a conjunction of equations $(x_1 = x_2)$ and then $[\Gamma]$ is equal to $[x = y]$.*
2. *Each relation in Γ is definable by a conjunction of inequalities $(x_1 \leq x_2)$ and then $[\Gamma]$ is equal to $[x \leq y]$.*
3. *Each relation in Γ is definable by the formula of the form $\bigwedge_{i=1}^{n}(x_{i_1} \leq x_{i_2} \vee \ldots \vee x_{i_1} \leq x_{i_k})$ and then provided Γ satisfies neither condition 1 nor 2, the set $[\Gamma]$ is equal to $[\Gamma_{Left}]$.*
4. *Each relation in Γ is definable by a formula of the form $\bigwedge_{i=1}^{n}(x_{i_2} \leq x_{i_1} \vee \ldots \vee x_{i_k} \leq x_{i_1})$ and then provided Γ satisfies neither condition 1 nor 2, the set $[\Gamma]$ is equal to $[\Gamma_{Right}]$.*
5. *Each relation in Γ is preserved by essentially unary surjective polymorphisms only.*

First, we want to distinguish $[\Gamma_{Left}]$ and $[\Gamma_{Right}]$ from each other and then from $[(x_1 \leq x_2)]$. Recall that in our language, see Theorem 4, sets $[\Gamma_1]$ and $[\Gamma_2]$ for some positive temporal relations Γ_1 and Γ_2 are different if there exists a function that preserves relations from exactly one of these sets.

Lemma 10. *The language Γ_{Left} is closed under dual-spp operation but it is not closed under spp. Dually, Γ_{Right} is closed under spp but it is not closed under dual-spp.*

It is quite obvious that $(x_1 \leq x_2)$ is closed under both *spp* and *dual-spp*. In [3], it is shown that $(x_1 \leq x_2)$ is closed under *median*, which is the ternary function that returns the median of its three argument. It is not hard to show that *median* is a surjective oligopotent QNU polymorphisms. To distinguish $(x_1 \leq x_2)$ from Γ_{Left} and Γ_{Right} we show that the last two relations are not closed under any surjective QNU polymorphism.

Lemma 11. *Neither Γ_{Left} nor Γ_{Right} is closed under any surjective oligopotent QNU polymorphism.*

It turns out that the relation defined by $(x_1 \leq x_2 \vee x_1 \leq x_3)$ has the same expressive power, in the sense of cp-definability, as Γ_{Left}. In the same context the relation defined by $(x_2 \leq x_1 \vee x_3 \leq x_1)$ is as powerful as the whole Γ_{Right}.

Lemma 12. *Every relation in Γ_{Left} has a cp-definition over $(x_1 \leq x_2 \vee x_1 \leq x_3)$, that is, $[(x_1 \leq x_2 \vee x_1 \leq x_3)] = [\Gamma_{Left}]$. Similarly, $[(x_2 \leq x_1 \vee x_3 \leq x_1)] = [\Gamma_{Right}]$.*

Consider the following forms of filters.

$$z_1 \approx \ldots \approx z_k \prec y_1 \tag{2}$$

$$z_1 \prec y_1 \approx \ldots \approx y_k \tag{3}$$

We say that a preorder $\preceq$ with domain $Dom(\preceq) = \{x_1, \ldots, x_n\}$ is of the form, let's say, (2) if $n = k + 1$ and $x_{i_1} \approx \ldots \approx x_{i_{n-1}} \prec x_{i_n}$ for some permutation of $\{x_1, \ldots, x_n\}$.

The next theorem shows us the difference between positive temporal languages with non-unary and only unary polymorphism. This difference is expressed using their filters.

Theorem 6. *Let Γ be a positive temporal language. Consider the following conditions.*

1. *All filters minimal with respect to $\ll$ in Γ are of the form (2).*
2. *All filters minimal with respect to $\ll$ in Γ are of the form (3).*

If neither of these conditions holds, then sPol(Γ) contains only essentially unary polymorphisms.

The proof of this theorem is presented in the next section. Here, we show that the expressive power of a positive temporal template satisfying item 1 of Theorem 6 is not higher than the one of $(x_1 \leq x_2 \vee x_1 \leq x_3)$. A similar statement concerning languages satisfying item 2 of Theorem 6 and $(x_2 \leq x_1 \vee x_3 \leq x_1)$ is also true.

Lemma 13. *Let Γ be a positive temporal language.*

1. *If all minimal with respect to $\ll$ filters in $\mathcal{F}(\Gamma)$ are of the form $z_1 \prec y_1$, then $[\Gamma] \subseteq [x_1 \leq x_2]$.*
2. *If all minimal with respect to $\ll$ filters in $\mathcal{F}(\Gamma)$ are of the form (2) and at least one of them has a domain of size greater than or equal to 3, then $[\Gamma] = [\Gamma_{Left}]$.*
3. *If all minimal with respect to $\ll$ filters in $\mathcal{F}(\Gamma)$ are of the form (3) and at least one of them has a domain of size greater than or equal to 3, then $[\Gamma] = [\Gamma_{Right}]$.*
4. *If $[\Gamma] \subseteq [x_1 \leq x_2]$, then either $[\Gamma] = [x_1 \leq x_2]$ or $[\Gamma] = [x_1 = x_2]$.*

Proof of Theorem 5. (Part One) First we show for a positive temporal language Γ that either $[\Gamma]$ is equal to exactly one of the following

1. $[x_1 \leq x_2 \vee x_1 \leq x_3]$
2. $[x_2 \leq x_1 \vee x_3 \leq x_1]$
3. $[x_1 \leq x_2]$
4. $[x_1 = x_2]$

or Γ is closed under essentially unary polymorphisms only. By Theorem 6, it is enough to prove that if all filters minimal with respect to $\ll$ of Γ are either of the form (2) or of the form (3), then $[\Gamma]$ is equal to exactly one of the above classes.

By Lemma 13 we obtain that $[\Gamma]$ is equal to at least one of them. To show that these classes are pairwise disjoint we use appropriate polymorphisms from the preceding lemmas and Theorem 4. By Lemma 10 we have that the first of the above families is closed under *dual-spp* operation but is not preserved by *spp*. By Lemma 11, it is not closed under any surjective oligopotent QNU polymorphism. Using the same lemmas we obtain a similar statement about the second family. Further, from [3] we know that the third family is closed under a surjective oligopotent QNU polymorphism. To distinguish the third and the fourth of the above sets using a function (in fact a permutation of rational numbers) note that the former is not closed under any strictly decreasing unary function.

(Part Two) Since all the classes are pairwise disjoint we can infer from Lemma 13 that $[\Gamma] = [\Gamma_{Left}]$ if and only if all filters from Γ are of the form (2) and at least one of them has a domain of size at least 3. Now, we show that if it is the case, then each relation from Γ is definable by a formula of the form $\bigwedge_{i=1}^{n}(x_{i_1} \leq x_{i_2} \vee \ldots \vee x_{i_1} \leq x_{i_k})$. Indeed it is enough to introduce a clause $(y_1 \leq y_2 \vee \ldots \vee y_1 \leq y_m)$ for each filter of the form $y_2 \approx \ldots \approx y_m \prec y_1$ from $\mathcal{F}(R)$. Now it is not hard to prove that a valuation q does not satisfy such a clause if and only if q is compatible with some preorder more general than $y_2 \approx \ldots \approx y_m \prec y_1$. Because it is a filter of R, by Lemma 8 we are done.

Similarly we can prove that if $[\Gamma] = [\Gamma_{Right}]$, then each relation from Γ may be defined by a formula of the form $\bigwedge_{i=1}^{n}(x_{i_2} \leq x_{i_1} \vee \ldots \vee x_{i_k} \leq x_{i_1})$.

Further, if $[\Gamma]$ is equal to $[x_1 \leq x_2]$, then all filters are of the form $z \prec y$. Here, we can show that each relation in Γ is definable by a conjunction of inequalities.

Finally, if $[\Gamma]$ is equal to $[x_1 = x_2]$, then, by Lemma 13, all filters are of the form $z \prec y$ and for each such a filter the set $\mathcal{F}(\Gamma)$ contains $y \prec z$ as well. Therefore it is not hard to show that each relation in Γ may be defined as a conjunction of equalities. □

6 Proof of Theorem 6

Although the last section contains the proof of Theorem 5, Theorem 6 was left without an explanation. This section is devoted to fill this hole.

The idea behind the proof is to show that if a positive temporal template Γ contains a filter that is not of the form (2) and a filter that is not of the form (3), then $[\Gamma]$ contains some positive non-negative equality relation R. By Corollary 2, the relation R and hence, by Theorem 4, the language Γ is closed only under essentially unary surjections.

To prove this we consider a few cases. In most of them, we use the following lemma.

If an n-ary positive temporal relation $R(x_1, \ldots, x_n)$ is different from $\mathbb{Q}^n$ and contains $\bigvee_{i \neq j} x_i = x_j$ for $1 \leq i, j \leq n$ as a subrelation then we call it potentially non-negative positive.

Lemma 14. *Let R be a potentially non-negative positive relation. Then it is closed under essentially unary polymorphism only.*

Consider the followings forms of filters.

$$x_1 \approx \ldots \approx x_k \prec y_1 \approx \ldots \approx y_l \tag{4}$$

$$x_1^1 \approx \ldots \approx x_{l_1}^1 \prec \ldots \prec x_1^n \approx \ldots \approx x_{l_n}^n \tag{5}$$

If $\mathcal{F}(\Gamma)$ contains a filter than is not of the form (2) and a filter that is not of the form (3), then one of the following cases holds.

1. The language Γ contains a filter of the form (4) where $k > 1$ and $l \geq 1$ as well as a filter of the same form where $l > 1$ and $k \geq 1$, or
2. there exists a filter of the form (5) for $n \geq 3$.

The next two sections handles these cases. The section 6.1 covers the first situation. The second case is taken care of by the section 6.2.

6.1 Filters of Range 2

Here we prove the following.

Proposition 1. *Let Γ be a positive temporal template with filters $\preceq_L$ and $\preceq_R$ defined as follows. The preorder $\preceq_L$ is of the form (4) with $k > 1$ and $l \geq 1$. The preorder $\preceq_R$ is also of the same form (4), but with $l > 1$ and $k \geq 1$. Then sPol(Γ) contains only essentially unary polymorphisms.*

Our strategy here is to show first that 'short' relations with 'short' filters of the form (4) with $k > 1$ and $l \geq 1$ are closed only under surjective unary polymorphisms or they express $(x_1 \leq x_2 \vee x_1 \leq x_3)$. Afterward, we consider arbitrary relations with arbitrary 'long' filters. For a 'long' relation R_L, we show that it can express a 'short' relation R_S. Therefore R_L can express everything expressible by R_S; equivalently, R_L cannot have more surjective polymorphisms than R_S. Lemmas 5, 4, and 9 ensure that R_S obtained from R_L has an appropriate filter. 'Short' relations have either arity 3 or 4. The first case is handled by Corollary 5, the second case by Lemma 17. See the example at the end of this subsection. First, we give two preliminary lemmas.

Lemma 15. *Let $R(x_1, x_2, x_3)$ be a positive temporal relation with a filter $x_1 \approx x_2 \prec x_3$. Moreover assume that $x_3 \prec x_1 \approx x_2$ does not belong to $\mathcal{F}(R)$. Then $(v_1 \leq v_2 \vee v_1 \leq v_3) \in [R]$.*

Lemma 16. *Let $R(x_1, x_2, x_3)$ be a positive temporal relation with filters $x_1 \approx x_2 \prec x_3$ and $x_3 \prec x_1 \approx x_2$. Then sPol$(\Gamma)$ contains only essentially unary polymorphisms.*

As an immediate consequence of lemmas 15 and 16 we get the following.

Corollary 5. *Let $R(x_1, x_2, x_3)$ be a positive temporal relation with a filter $x_1 \approx x_2 \prec x_3$. Then either $(v_1 \leq v_2 \vee v_1 \leq v_3) \in [\Gamma]$ or sPol(Γ) contains only essentially unary polymorphisms.*

Lemma 17. *Let $R(x_1, x_2, x_3, x_4)$ be a positive temporal relation with a filter $x_1 \approx x_2 \approx x_3 \prec x_4$. Then either $(v_1 \leq v_2 \vee v_1 \leq v_3) \in [R]$ or sPol(R) contains only essentially unary polymorphisms.*

Lemma 18. *Let $R(x_1, \ldots, x_n)$ be a positive temporal relation. If R has any filter of the form (4) where $k > 1$ and $l \geq 1$, then either $(v_1 \leq v_2 \vee v_1 \leq v_3) \in [R]$ or sPol(Γ) contains only essentially unary polymorphisms.*

Similarly, we can show that if a positive temporal relation has any filter of the form (4) where $k \geq 1$ and $l > 1$, then either it is closed only under essentially unary surjective polymorphisms or it can express $(x_2 \leq x_1 \vee x_3 \leq x_1)$. The following statement in fact ends the proof of Proposition 1.

Lemma 19. *If both $(v_1 \leq v_2 \vee v_1 \leq v_3)$ and $(v_2 \leq v_1 \vee v_3 \leq v_1)$ belong to $[\Gamma]$, then Γ is closed only under essentially unary polymorphisms.*

Proof of Proposition 1. Let R be a relation of Γ with $\preceq_L$. Then, by Lemma 18, either $(v_1 \leq v_2 \vee v_1 \leq v_3) \in [R]$ or sPol(R) contains only essentially unary polymorphisms. Similarly, for some R_1 that has $\preceq_R$ as a filter we can show that either all its surjective polymorphisms are essentially unary or $(v_2 \leq v_1 \vee v_3 \leq v_1)$ belongs to $[R_1]$. To complete the proof we use Lemma 19. □

We finish this subsection with an example that illustrates Proposition 1.

Example 4. Consider a positive relation R_L given by $(x_1 \leq x_2 \vee x_1 \leq x_3) \wedge (x_5 \leq x_4 \vee x_6 \leq x_4) \wedge \phi(y_1, \ldots, y_m)$ where $\{x_1, \ldots, x_6\} \cap \{y_1, \ldots, y_m\} = \emptyset$. It is straightforward to show that $x_2 \approx x_3 \prec x_1$ as well as $x_4 \prec x_5 \approx x_6$ are minimal wrt $\ll$ filters of R – see Example 3. We now claim that sPol(Γ) contains essentially unary polymorphisms only or equivalently, by Corollary 3, the problem *QCSP*(Γ) is NP-hard.

Now, define R_S as $\exists y_1 \ldots \exists y_m \exists x_4 \exists x_5 \exists x_6 R_L(x_1, \ldots, x_6, y_1, \ldots, y_m)$ By Lemma 5, we have that R_S inherits the filter $x_2 \approx x_3 \prec x_1$. From Lemma 15 we infer that $(v_1 \leq v_2 \vee v_1 \leq v_3)$ belongs to $[R_S]$. Hence it belongs to $[R_L]$; note that R_S is cp-definable in R_L – recall the definition of R_S above. Similarly we can show that $(v_2 \leq v_1 \vee v_3 \leq v_1) \in R_L$. By Lemma 19 we have that sPol(R_L) contains essentially unary polymorphisms only.

6.2 Filters of Range Greater Than 2

What remains to prove is the following.

Proposition 2. *Let Γ be a positive temporal language. If there exists any minimal with respect to $\ll$ filter in $\mathcal{F}(\Gamma)$ whose range is strictly greater than 2, then sPol(Γ) contains only essentially unary polymorphisms.*

The strategy here is similar to one in the preceding section. We express 'short' relations with 'short' filters using 'long' relations with 'long' filters and show that 'short' relations are closed under essentially unary surjections only. Here, 'short' filters are of the form (5) where each $l_i = 1$ for all $1 \leq i \leq n$. In turn, we think that a relation R_S is 'short' if it contains a 'short' filter $\preceq_S$ and $Dom(R_S) = Dom(\preceq_S)$. Each 'long' relation has at least one (arbitrary) filter of the form (5). For example, the filter $x_1 \approx x_2 \prec x_3 \prec x_4$ is 'long', but the filter $x_1 \prec x_3 \prec x_4$ is 'short'.

7 Proof of Theorem 2

In [3] it is shown that each positive temporal language Γ from case 2 is decidable in NLOGSPACE. To prove Theorem 1 we need hardness as well.

Lemma 20. *Let Γ be a positive temporal language such that each its relation is definable as a conjunction of weak inequalities but not as a conjunction of equalities. Then $QCSP(\Gamma)$ is NLOGSPACE-complete.*

Proof. (of Theorem 2) By Theorem 5 we have that each positive temporal Γ is either definable as in one of the conditions 1–4 of Theorem 1 or it is closed under essentially unary surjections only. Lemma 20 gives us the complexity characterization of item 2. Item 1 is characterized in [4]. In [10] we give the complexity proof for items 3 and 4. Finally, by Corollary 3, we have that for any other positive temporal language Γ the problem $QCSP(\Gamma)$ is NP-hard. □

Acknowledgements. We thank Jerzy Marcinkowski for turning our attention to [4].

References

1. Allen, J.F.: Maintaining knowledge about temporal intervals. Commun. ACM 26(11), 832–843 (1983)
2. Bodirsky, M.: Constraint Satisfaction Problems with Infinite Domains. PhD thesis, Humboldt-Universität zu Berlin (2004), http://www2.informatik.hu-berlin.de/ bodirsky/publications/diss.html
3. Bodirsky, M., Chen, H.: Qualitative temporal and spatial reasoning revisited. In: Duparc, J., Henzinger, T.A. (eds.) CSL 2007. LNCS, vol. 4646. Springer, Heidelberg (2007)
4. Bodirsky, M., Chen, H.: Quantified equality constraints. In: 22nd IEEE Symposium on Logic in Computer Science (LICS 2007), Proceedings. IEEE Computer Society Press, Los Alamitos (2007)
5. Bodirsky, M., Kára, J.: A fast algorithm and lower bound for temporal reasonning, http://www2.informatik.hu-berlin.de/bodirsky/en/publications.php
6. Bodirsky, M., Kára, J.: The complexity of equality constraint languages. In: Grigoriev, D., Harrison, J., Hirsch, E.A. (eds.) CSR 2006. LNCS, vol. 3967, pp. 114–126. Springer, Heidelberg (2006)
7. Bodirsky, M., Kára, J.: The complexity of temporal constraint satisfaction problems. In: Ladner, R.E., Dwork, C. (eds.) Proceedings of the 40th Annual ACM Symposium on Theory of Computing, pp. 29–38. ACM, New York (2008)

8. Boerner, F., Bulatov, A., Jeavons, P., Krokhin, A.: Quantified constraints: Algorithms and complexity. In: Proceedings of CSL and the 8th Kurt Gödel Colloquium. LNCS. Springer, Heidelberg
9. Bulatov, A.A.: A dichotomy theorem for constraints on a three-element set. In: Proceedings 43rd IEEE Symposium on Foundations of Computer Science (FOCS 2002), pp. 649–658 (2002)
10. Charatonik, W., Wrona, M.: Quantified positive temporal constraints. In: Kaminski, M., Martini, S. (eds.) CSL 2008. LNCS, vol. 5213, pp. 94–108. Springer, Heidelberg (2008)
11. Cohen, D., Jeavons, P.: The complexity of constraints languages. In: Rossi, F., van Beek, P., Walsh, T. (eds.) Handbook of Constraint Programming. Elsevier, Amsterdam (2006)
12. Feder, T., Vardi, M.Y.: Monotone monadic SNP and constraint satisfaction. In: Proceedings of 25th ACM Symposium on the Theory of Computing (STOC), pp. 612–622 (1993)
13. Fisher, M., Gabbay, D., Vila, L.: Handbook of Temporal Reasoning in Artificial Intelligence. Elsevier, Amsterdam (2005)
14. Hodges, W.: A shorter model theory. Cambridge University Press, Cambridge (1997)
15. Jeavons, P.G., Cohen, D.A., Gyssens, M.: Closure properties of constraints. Journal of the ACM 44, 527–548 (1997)
16. Krokhin, A., Jeavons, P., Jonsson, P.: A complete classification of complexity in Allens algebra in the presence of a non-trivial basic relation. In: Proceedings of the 17th International Joint Conference on Artificial Intelligence, pp. 83–88. Morgan Kaufmann, San Francisco (2001)
17. Möhring, R.H., Skutella, M., Stork, F.: Scheduling with and/or precedence constraints. SIAM J. Comput. 33(2), 393–415 (2004)
18. Renz, J., Nebel, B.: Qualitative spatial reasoning using constraint calculi. In: Aiello, M., Pratt-Hartmann, I., van Benthem, J. (eds.) Handbook of Spatial Logics. Springer, Heidelberg (2007)
19. Schaefer, T.J.: The complexity of satisfiability problems. In: Proceedings 10th ACM Symposium on Theory of Computing, STOC 1978, pp. 216–226 (1978)

A Logic of Singly Indexed Arrays*

Peter Habermehl[1], Radu Iosif[2], and Tomáš Vojnar[3]

[1] LSV, ENS Cachan, CNRS, INRIA; 61 av. du Président Wilson, F-94230 Cachan, France and LIAFA, University Paris 7, Case 7014, 75205 Paris Cedex 13
haberm@liafa.jussieu.fr
[2] VERIMAG, CNRS, 2 av. de Vignate, F-38610 Gières, France
iosif@imag.fr
[3] FIT BUT, Božetěchova 2, CZ-61266, Brno, Czech Republic
vojnar@fit.vutbr.cz

Abstract. We present a logic interpreted over integer arrays, which allows difference bound comparisons between array elements situated within a constant sized window. We show that the satisfiability problem for the logic is undecidable for formulae with a quantifier prefix $\{\exists,\forall\}^*\forall^*\exists^*\forall^*$. For formulae with quantifier prefixes in the $\exists^*\forall^*$ fragment, decidability is established by an automata-theoretic argument. For each formula in the $\exists^*\forall^*$ fragment, we can build a flat counter automaton with difference bound transition rules (FCADBM) whose traces correspond to the models of the formula. The construction is modular, following the syntax of the formula. Decidability of the $\exists^*\forall^*$ fragment of the logic is a consequence of the fact that reachability of a control state is decidable for FCADBM.

1 Introduction

Arrays are commonplace data structures in most programming languages. Reasoning about programs with arrays calls for expressive logics capable of encoding pre- and post-conditions as well as loop invariants. Moreover, in order to automate program verification, one needs tractable logics whose satisfiability problems can be answered by efficient algorithms.

In this paper, we present a logic of integer arrays based on universally quantified comparisons between array elements situated within a constant sized window, i.e., quantified boolean combinations of basic formulae of the form $\forall i \,.\, \gamma(i) \rightarrow a_1[i+k_1] - a_2[i+k_2] \leq m$ where γ is a positive boolean combination of bound and modulo constraints on the index variable i, a_1 and a_2 are array symbols, and $k_1, k_2, m \in \mathbb{Z}$ are integer constants. Hence the name of Single Index Logic (**SIL**). Note that **SIL** can also be viewed as a fragment of Presburger arithmetic extended with uninterpreted functions mapping naturals to integers.

The main idea in defining the logic is that only one universally quantified index may be used on the right hand side of the implication within a basic formula. According to [10], this restriction is not a real limitation of the expressive power of the logic

* The work was supported by the French Ministry of Research (RNTL project AVERILES), the Czech Grant Agency (projects 102/07/0322, 102/05/H050), the Czech-French Barrande project MEB 020840, and the Czech Ministry of Education by project MSM 0021630528.

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 558–573, 2008.

since a formula using two or more universally quantified variables in a difference bound constraint on array values can be equivalently written in the form above, by introducing fresh array symbols. This technique has been detailed in [10].

Working directly with singly-indexed formulae allows to devise a simple and efficient decision procedure for the satisfiability problem of the $\exists^*\forall^*$ fragment of **SIL**, based on a modular translation of formulae into deterministic flat counter automata with difference bound transition rules (FCADBM). This is possible due to the fact that deterministic FCADBM are closed under union, intersection and complement, when considering their sets of traces.

The satisfiability problem for $\exists^*\forall^*$-**SIL** is thus reduced to checking reachability of a control state in an FCADBM. The latter problem has been shown to be decidable first in [6], by reduction to the satisfiability problem of Presburger arithmetic. Later on, the method described in [4] reduced this problem to checking satisfiability of a linear Diophantine system, leading to the implementation of the FLATA toolset [7].

Universally quantified formulae of the form $\forall i \,.\, \gamma(i) \rightarrow \upsilon(i)$ are a natural choice when reasoning about arrays as one usually tend to describe facts that must hold for all array elements (array invariants). A natural question is whether a more complex quantification scheme is possible, while preserving decidability. In this paper, we show that the satisfiability problem for the class of formulae with quantifier prefixes of the form $\forall^*\exists^*\forall^*$ is already undecidable, providing thus a formal reason for the choice of working with existentially quantified boolean combinations of universal basic formulae. The contribution of this paper is hence three-fold:

- we show that the satisfiability problem for the class of formulae with quantifier prefixes of the form $\forall^*\exists^*\forall^*$ is undecidable,
- we define a class of counter automata that is closed under union, intersection and complement of their sets of traces,
- we provide a decision procedure for the satisfiability problem within the fragment of formulae with alternation depth of at most one, based on a modular, simple, and efficient translation of formulae into counter automata.

The practical usefulness of the **SIL** logic is shown by giving a number of examples of properties that are recurrent in programs handling array data structures.

Related Work. The saga of papers on logical theories of arrays starts with the seminal paper [15], in which the read and write functions from/to arrays and their logical axioms were introduced. A decision procedure for the quantifier-free fragment of the theory of arrays was presented in [12]. Since then, various quantifier-free decidable logics on arrays have been considered—e.g., [17,13,11,16,1,8].

In [5], an interesting logic, within the $\exists^*\forall^*$ quantifier fragment, is developed. Unlike our decision procedure based on automata theory, the decision procedure of [5] is based on a model-theoretic observation, allowing to replace universal quantification by a finite conjunction. The decidability of their theory depends on the decidability of the base theory of array values. However, compared to our results, [5] does not allow modulo constraints (allowing to speak about periodicity in the array values) nor reasoning about array entries at a fixed distance (i.e., reasoning about $a[i]$ and $a[i+k]$ for a constant k and a universally quantified index i). The authors of [5] give also interesting undecidability results for extensions of their logic. For example, they show that relating

adjacent array values ($a[i]$ and $a[i+1]$), or having nested reads, leads to undecidability. In their setting, undecidability occurs as a consequence of allowing disjunctions between predicates involving array value terms ($a[i]$). We circumvent this problem by forbidding disjunctions on the right hand side of the implication, within universally quantified formulae of the form $\forall i \,.\, \gamma(i) \rightarrow \upsilon(i)$.

A restricted form of universal quantification within $\exists^*\forall^*$ formulae is also allowed in [2], where decidability is obtained based on a small model property. Unlike [5] and our work, [2] allows a hierarchy-restricted form of array nesting. However, similar to the restrictions presented above, neither modulo constraints on indices, nor reasoning about array entries at a fixed distance are allowed. A similar restriction not allowing to express properties of consecutive elements of arrays appears also in [3], where a quite general $\exists^*\forall^*$ logic on multisets of elements with associated data values is considered.

The closest in spirit to the present paper is our previous work in [10]. There, we established decidability of formulae in the $\exists^*\forall^*$ quantifier prefix class when references to adjacent array values (e.g., $a[i]$ and $a[i+1]$) are not used in disjunctive terms. However, there are two essential differences between this work and the one reported in [10].

On one hand, the basic propositions from [10], allowing multiple universally quantified indices could not be translated directly into counter automata. This led to a complex elimination procedure based on introducing new array symbols, which produces singly-indexed formulae. However, the automata resulting from this procedure are not closed under complement. Therefore, negation had to be eliminated prior to reducing the formula to the singly-indexed form, causing further complexity. In the present work, we start directly with singly-indexed formulae, convert them into automata, and compose the automata directly using boolean operators (union, intersection, complement).

On the other hand, using universally quantified array property formulae as building blocks for the formulae, although intuitive, is not formally justified in [10]. Here, we prove that alternating quantifiers to a depth more than two leads to undecidability.

Roadmap. The paper is organised as follows. Section 2 introduces the necessary notions on counter automata and defines the class of FCADBM. Section 3 defines the logic **SIL**. Next, Section 4 gives the undecidability result for the entire logic, while Section 5 proves decidability of the satisfiability for the $\exists^*\forall^*$ fragment, by translation to deterministic FCADBM. Finally, Section 6 presents some concluding remarks. For space reasons, most of the proofs are deferred to [9].

2 Counter Automata

Given a formula φ, we denote by $FV(\varphi)$ the set of its free variables. If we denote a formula as $\varphi(x_1,...,x_n)$, we assume $FV(\varphi) \subseteq \{x_1,...,x_n\}$. For $\varphi(x)$, we denote by $\varphi[t/x]$ the formula in which each free occurrence of x is replaced by a term t. Given a formula φ, we denote by $\models \varphi$ the fact that φ is logically valid, i.e., it holds in every structure corresponding to its signature.

A *difference bound matrix* (DBM) formula is a conjunction of inequalities of the forms (1) $x - y \leq c$, (2) $x \leq c$, or (3) $x \geq c$, where $c \in \mathbb{Z}$ is a constant. We denote by $\top$ (true) the empty DBM. It is well-known that the negation of a DBM formula is

equivalent to a finite disjunction of *pairwise disjoint* DBM formulae since, e.g., $\neg(x - y \leq c) \iff y - x \leq -c - 1$ and $\neg(x \leq c) \iff x \geq c + 1$. In particular, the negation of $\top$ is the empty disjunction, denoted as $\bot$ (false).

A *counter automaton* (CA) is a tuple $A = \langle \mathbf{x}, Q, I, \rightarrow, F \rangle$ where:

- $\mathbf{x}$ is a finite set of counters ranging over $\mathbb{Z}$,
- Q is a finite set of control states,
- $I \subseteq Q$ is a set of initial states,
- $\rightarrow$ is a transition relation given by a set of rules $q \xrightarrow{\varphi(\mathbf{x},\mathbf{x}')} q'$ where φ is an arithmetic formula relating current values of counters $\mathbf{x}$ to their future values $\mathbf{x}' = \{x' \mid x \in \mathbf{x}\}$,
- $F \subseteq Q$ is a set of final states.

A *configuration* of a counter automaton A is a pair (q, ν) where $q \in Q$ is a control state, and $\nu : \mathbf{x} \rightarrow \mathbb{Z}$ is a valuation of the counters in $\mathbf{x}$. For a configuration $c = (q, \nu)$, we designate by $val(c) = \nu$ the valuation of the counters in c. A configuration (q', ν') is an *immediate successor* of (q, ν) if and only if A has a transition rule $q \xrightarrow{\varphi(\mathbf{x},\mathbf{x}')} q'$ such that $\models \varphi(\nu(\mathbf{x}), \nu'(\mathbf{x}'))$. A configuration c is a *successor* of another configuration c' if and only if there exists a finite sequence of configurations $c = c_1 c_2 \ldots c_n = c'$ such that, for all $1 \leq i < n$, c_{i+1} is an immediate successor of c_i. Given two control states $q, q' \in Q$, a run of A from q to q' is a finite sequence of configurations $c_1 c_2 \ldots c_n$ with $c_1 = (q, \nu)$, $c_n = (q', \nu')$ for some valuations $\nu, \nu' : \mathbf{x} \rightarrow \mathbb{Z}$, and c_{i+1} is an immediate successor of c_i, for all $1 \leq i < n$. Let $\mathcal{R}(A)$ denote the set of runs of A from some initial state $q_0 \in I$ to some final state $q_f \in F$, and $Tr(A) = \{val(c_1)val(c_2)\ldots val(c_n) \mid c_1 c_2 \ldots c_n \in \mathcal{R}(A)\}$ be its set of *valuation traces*. If $\mathbf{z} \subseteq \mathbf{x}$ is a subset of the counters of A and $\nu : \mathbf{x} \rightarrow \mathbb{Z}$ is a valuation of its counters, let $\nu\downarrow_{\mathbf{z}}$ be the restriction of ν to the counters in $\mathbf{z}$. If $c = (q, \nu)$ is a configuration of A, we denote $c\downarrow_{\mathbf{z}} = (q, \nu\downarrow_{\mathbf{z}})$ and $Tr(A)\downarrow_{\mathbf{z}} = \{val(c_1)\downarrow_{\mathbf{z}} val(c_2)\downarrow_{\mathbf{z}} \ldots val(c_n)\downarrow_{\mathbf{z}} \mid c_1 c_2 \ldots c_n \in \mathcal{R}(A)\}$.

A counter $z \in \mathbf{x}$ is called a *parameter* of A if and only if, for each $\sigma = \nu_1 \ldots \nu_n \in Tr(A)$, we have $\nu_1(z) = \ldots = \nu_n(z)$, in other words the value of the counter does not change during any run of A.

A *control path* in a counter automaton A is a finite sequence $q_1 q_2 \ldots q_n$ of control states such that, for all $1 \leq i < n$, there exists a transition rule $q_i \xrightarrow{\varphi_i} q_{i+1}$. A *cycle* is a control path starting and ending in the same control state. An *elementary cycle* is a cycle in which each state appears only once, except for the first one, which appears both at the beginning and at the end. A counter automaton is said to be *flat* iff each control state belongs to at most one elementary cycle.

A counter automaton A is said to be *deterministic* if and only if (1) it has exactly one initial state, and (2) for each pair of transition rules with the same source state $q \xrightarrow{\varphi} q'$ and $q \xrightarrow{\psi} q''$, we have $\models \neg(\varphi \wedge \psi)$. It is easy to prove that, given a deterministic counter automaton A, for each sequence of valuations $\nu_1 \nu_2 \ldots \nu_n \in Tr(A)$ there exists exactly one control path $q_1 q_2 \ldots q_n$ such that $(q_0, \nu_1)(q_1, \nu_2)\ldots(q_{n-1}, \nu_n) \in \mathcal{R}(A)$.

2.1 Flat Counter Automata with DBM Transition Rules

In the rest of the paper, we use the class of *flat counter automata with DBM transition rules* (FCADBM). They are defined to be flat counter automata where each transition

in a cycle is labelled by a DBM formula and each transition not in a cycle is labelled by a conjunction of a DBM formula with a (possibly empty) conjunction of modulo constraints on parameters of the form $z \equiv_s t$ where $0 \leq t < s$.

An extension of this class has been studied in [10]. Using results of [6,4], [10] shows that, given a CA $A = \langle \mathbf{x}, Q, I, \rightarrow, F \rangle$ in the class it considers and a pair of control states $q, q' \in Q$, the set $V_{q,q'} = \{(\nu, \nu') \in (\mathbf{x} \mapsto \mathbb{Z})^2 \mid A \text{ has a run from } (q, \nu) \text{ to } (q', \nu')\}$ is Presburger-definable. As an immediate consequence, the emptiness problem for A, i.e., $Tr(A) \stackrel{?}{=} \emptyset$, is decidable.

Theorem 1. *The emptiness problem for FCADBM is decidable.*

In this section, we show that *deterministic* FCADBM are closed under union, intersection, and complement of their sets of traces. Let $A_i = \langle \mathbf{x}, Q_i, \{q_{0i}\}, \rightarrow_i, F_i \rangle$, $i = 1, 2$, be two deterministic FCADBM with the same set of counters. Note that this is not a restriction as one can add unrestricted counters without changing the behaviour of a CA. We first show closure under intersection by defining the CA $A_1 \otimes A_2 = \langle \mathbf{x}, Q_1 \times Q_2, \{(q_{01}, q_{02})\}, \rightarrow, F_1 \times F_2 \rangle$ where $(q_1, q_2) \xrightarrow{\varphi} (q_1', q_2') \iff q_1 \xrightarrow{\psi_1}_1 q_1',\ q_2 \xrightarrow{\psi_2}_2 q_2'$, and $\models \varphi \leftrightarrow \psi_1 \wedge \psi_1$. The next lemma proves the correctness of our construction.

Lemma 1. *For any two deterministic FCADBM $A_i = \langle \mathbf{x}, Q_i, \{q_{i0}\}, \rightarrow_i, F_i \rangle$, $i = 1, 2$, $A_1 \otimes A_2$ is a deterministic FCADBM, and $Tr(A_1 \otimes A_2) = Tr(A_1) \cap Tr(A_2)$.*

Let $A = \langle \mathbf{x}, Q, I, \rightarrow, F \rangle$ be a deterministic FCADBM. Then we define $\overline{A} = \langle \mathbf{x}, Q \cup \{q_s\}, I, \rightarrow', (Q \setminus F) \cup \{q_s\} \rangle$ where $q_s \notin Q$ is a fresh sink state. The transition relation $\rightarrow'$ is defined as follows. For a control state $q \in Q$, let $O_A(q) = \bigvee_{q \xrightarrow{\varphi} q'} \varphi$.[1] Then, we have:

- $q_s \xrightarrow{\top}' q_s$, $q \xrightarrow{\varphi}' q'$ for each $q \xrightarrow{\varphi} q'$, and
- $q \xrightarrow{\psi_i}' q_s$, for all $1 \leq i \leq k$, where ψ_i are (unique) conjunctions of DBMs and modulo constraints[2] such that $\models \neg O_A(q) \leftrightarrow \bigvee_{i=1}^{k} \psi_i$ and $\models \neg(\psi_i \wedge \psi_j)$ for $i \neq j$, $1 \leq i, j \leq k$.

Flatness of $\overline{A}$ is a consequence of the fact that the only cycle of $\overline{A}$, which did not exist in A, is the self-loop around q_s. That is, the newly added transitions do not create new cycles. It is immediate to see that $\overline{A}$ is deterministic whenever A is. The following lemma formalises correctness of the complement construction, proving thus that deterministic FCADBM are effectively closed under union[3], intersection, and complement of their sets of traces.

Lemma 2. *Given a deterministic FCADBM $A = \langle \mathbf{x}, Q, \{q_0\}, \rightarrow, F \rangle$, for any finite sequence of valuations $\sigma \in (\mathbf{x} \mapsto \mathbb{Z})^*$, we have $\sigma \in Tr(A)$ if and only if $\sigma \notin Tr(\overline{A})$.*

[1] If q has no immediate successors, then $O_A(q)$ is false by default.

[2] The negation of $z \equiv_s t$ with $t < s$ is equivalent to $\bigvee_{t' \in \{0,\ldots,s-1\} \setminus \{t\}} z \equiv_s t'$.

[3] The FCADBM whose set of traces is the union of the sets of traces of two given FCADBM A_1, A_2 can be obtained simply as $\overline{\overline{A_1} \otimes \overline{A_2}}$.

3 A Logic of Integer Arrays

3.1 Syntax

We consider three types of variables. The *array-bound variables* (k,l) appear within the bounds that define the intervals in which some property is required to hold. Let *BVar* denote the set of *array-bound variables*. The *index* (i,j) and *array* (a,b) *variables* are used in array terms. Let *IVar* denote the set of *index variables* and *AVar* denote the set of *array variables*. All variable sets are supposed to be finite and of known cardinality.

Fig. 1 shows the syntax of the Single Index Logic **SIL**. The term $|a|$ denotes the length of an array variable a. We use the symbol $\top$ to denote the boolean value *true*. In the following, we will write $f \leq i \leq g$ instead of $f \leq i \ \wedge \ i \leq g$, $i < f$ instead of $i \leq f-1$, $i = f$ instead of $f \leq i \leq f$, $\varphi_1 \vee \varphi_2$ instead of $\neg(\neg\varphi_1 \wedge \neg\varphi_2)$, and $\forall i \,.\, \upsilon(i)$ instead of $\forall i \,.\, \top \rightarrow \upsilon(i)$. If $\ell_1(k_1),\ldots,\ell_n(k_n)$ are array-bound terms with free variables $k_1,\ldots,k_n \in BVar$, respectively, we write any DBM formula φ on terms $a_1[\ell_1],\ldots,a_n[\ell_n]$, as a shorthand for $(\bigwedge_{k=1}^{n} \forall j \,.\, j = \ell_k \rightarrow a_k[j] = l_k) \wedge \varphi[l_1/a_1[\ell_1],\ldots,l_n/a_n[\ell_n]]$, where $l_1,\ldots,l_n$ are fresh array-bound variables.

$n,m,p\ldots$	$\in$	$\mathbb{Z}$	constants
$k,l,\ldots$	$\in$	*BVar*	array-bound variables
$i,j,\ldots$	$\in$	*IVar*	index variables
$a,b,\ldots$	$\in$	*AVar*	array variables
$\sim$	$\in$	$\{\leq,\geq\}$	
B	$:=$	$n \mid k+n \mid \lvert a\rvert+n$	array-bound terms
G	$:=$	$\top \mid i-j \leq n \mid i \leq B \mid B \leq i \mid i \equiv_s t \mid G \wedge G \mid G \vee G$	guard expressions $(0 \leq t < s)$
V	$:=$	$a[i+n] \sim B \mid a[i+n]-b[i+m] \sim p \mid$ $i-a[i+n] \sim m \mid V \wedge V$	value expressions
C	$:=$	$B \sim n \mid B-B \leq n \mid B \equiv_s t$	array-bound constraints $(0 \leq t < s)$
P	$:=$	$\forall i \,.\, G \rightarrow V$	array properties
F	$:=$	$P \mid C \mid \neg F \mid F \ \wedge \ F \mid F \ \vee \ F \mid \exists i \,.\, F$	formulae

Fig. 1. Syntax of the Single Index Logic

For reasons that will be made clear later on, we allow only one index variable to occur within the right hand side of the implication in an array property formula $\forall i \,.\, \gamma \rightarrow \upsilon$, i.e., we require $FV(\upsilon) \cap IVar = \{i\}$. Hence the name Single Index Logic (**SIL**). Note that this does not restrict the expressive power w.r.t. the logic considered in [10]. One can always circumvent this restriction by using the method from [10] based on adding new array symbols together with a transitive (increasing, decreasing, or constant) constraint on their adjacent values. This way a relation between arbitrarily distant entries $a[i]$ and $b[j]$ is decomposed into a sequence of relations between neighbouring entries of a, b, and entries of the auxiliary arrays. However this transformation would greatly complicate the decision procedure, hence we prefer to avoid it here.

Notice also that one can compare an array value with an array-bound variable, or with another array value on the right hand side of an implication in an array property formula $\forall i \,.\, \gamma \rightarrow \upsilon$, but one cannot relate two or more array values with array-bound

parameters in the same expression. Allowing more complex comparisons between array values would impact upon the decidability result reported in Section 5. For the same reason, disjunctive terms are not allowed on the right hand side of implications in array properties: allowing disjunctions in value expressions makes it possible to write a **SIL** formula that encodes all executions of a 2-counter machine with nested control structure (as shown already in [10]).

Let υ be a value expression written in the syntax of Fig. 1 (starting with the V non-terminal). Let $\mathcal{B}(\upsilon)$ be the formula defined inductively on the structure of υ as follows:

- $\mathcal{B}(a[i+n] \leq B) = \mathcal{B}(B \leq a[i+n]) = 0 \leq i+n < |a|$
- $\mathcal{B}(i - a[i+n] \leq m) = \mathcal{B}(a[i+n] - i \leq m) = 0 \leq i+n < |a|$
- $\mathcal{B}(a[i+n] - b[i+m] \leq p) = 0 \leq i+n < |a| \ \wedge \ 0 \leq i+m < |b|$
- $\mathcal{B}(\upsilon_1 \wedge \upsilon_2) = \mathcal{B}(\upsilon_1) \wedge \mathcal{B}(\upsilon_2)$

Intuitively, $\mathcal{B}(\upsilon)$ is the conjunction of all sanity conditions needed in order for the array accesses in υ to occur within proper bounds.

3.2 Semantics

Let us fix $AVar = \{a_1, a_2, \ldots, a_k\}$ as the set of array variables for the rest of this section. A *valuation* is a pair of partial functions $\langle \iota, \mu \rangle$ where $\iota : BVar \cup IVar \to \mathbb{Z}_\perp$ associates an integer value with every free integer variable, and $\mu : AVar \to \mathbb{Z}^*$ associates a *finite sequence* of integers with every array symbol $a \in AVar$. If $\sigma \in \mathbb{Z}^*$ is such a sequence, we denote by $|\sigma|$ its length and by σ_i its i-th element.

By $I_{\iota,\mu}(t)$, we denote the value of the term t under the valuation $\langle \iota, \mu \rangle$. The semantics of a formula φ is defined in terms of the forcing relation $\models$ as follows:

$$\begin{array}{rcl} I_{\iota,\mu}(|a|) & = & |\mu(a)| \\ I_{\iota,\mu}(a[i+n]) & = & \mu(a)_{\iota(i)+n} \end{array}$$

$$\begin{array}{rcl} \langle \iota, \mu \rangle \models a[i+n] \leq B & \iff & I_{\iota,\mu}(a[i+n]) \leq \iota(B) \\ \langle \iota, \mu \rangle \models A_1 - A_2 \leq n & \iff & I_{\iota,\mu}(A_1) - I_{\iota,\mu}(A_2) \leq n \\ \langle \iota, \mu \rangle \models \forall i \,.\, G \to V & \iff & \forall\, n \in \mathbb{Z} \,.\, \langle \iota[i \leftarrow n], \mu \rangle \models G \wedge \mathcal{B}(V) \to V \\ \langle \iota, \mu \rangle \models \exists i \,.\, F & \iff & \langle \iota[i \leftarrow n], \mu \rangle \models F \text{ for some } n \in \mathbb{N} \end{array}$$

Notice that the semantics of an array property formula $\forall i \,.\, G \to V$ ignores all values of i for which the array accesses of V are undefined since we consider only the values of i from $\mathbb{Z}$ that satisfy the safety assumption $\mathcal{B}(V)$. For space reasons, we do not give here a full definition of the semantics. However, the missing rules are standard in first-order arithmetic. A *model* of a **SIL** formula $\varphi(\mathbf{k}, \mathbf{a})$ is a valuation $\langle \iota, \mu \rangle$ such that the formula obtained by interpreting each variable $k \in \mathbf{k}$ as $\iota(k)$ and each array variable $a \in \mathbf{a}$ as $\mu(a)$ is logically valid: $\langle \iota, \mu \rangle \models \varphi$. We define $[\![\varphi]\!] = \{\langle \iota, \mu \rangle \mid \langle \iota, \mu \rangle \models \varphi\}$. A formula is said to be:

- *satisfiable* if and only if $[\![\varphi]\!] \neq \emptyset$, and
- *valid* if and only if $[\![\varphi]\!] = (BVar \cup IVar \to \mathbb{Z}_\perp) \times (AVar \to \mathbb{Z}^*)$

With these definitions, the *satisfiability problem* asks, given a formula φ if it has at least one model. Without losing generality, for the satisfiability problem, we can assume that the quantifier prefix of φ (in prenex normal form) does not start with $\exists$. Dually, the *validity problem* asks whether a given formula holds on every possible model. Symmetrically, for the validity problem, one can assume w.l.o.g. that the quantifier prefix of the given formula does not start with $\forall$.

3.3 Examples

We now illustrate the syntax, semantics, and use of the logic **SIL** on a number of examples. For instance, the formula $\forall i \,.\, a[i] = 0$ is satisfied by all functions μ mapping a to a finite sequence of 0's, i.e., $\mu(a) \in 0^*$. It is semantically equivalent to $\forall i \,.\, 0 \leq i < |a| \rightarrow a[i] = 0$, in which the range of i has been made explicit.

The formula $\forall i \,.\, 0 \leq i < k \rightarrow a[i] = 0$ is satisfied by all pairs $\langle \iota, \mu \rangle$ where μ maps a to a sequence whose first $\iota(k)$ elements (if they exist) are 0, i.e., $\mu(a) \in \{0^n \mid 1 \leq n < \iota(k)\} \cup 0^{\iota(k)}\mathbb{Z}^*$. It is semantically equivalent to $\forall i \,.\, 0 \leq i < \min(|a|, k) \rightarrow a[i] = 0$.

The capability of **SIL** to relate array entries at fixed distances (missing in many decidable logics such as those considered in [2,5,3]) is illustrated on a bigger example below. The modulo constraints on the index variables can then be used to state periodic facts. For instance, the formula $\forall i \,.\, i \equiv_2 0 \rightarrow a[i] = 0 \wedge \forall i \,.\, i \equiv_2 1 \rightarrow a[i] = 1$ describes the set of arrays a in which the elements on even positions have the value 0, and the elements on odd positions have the value 1.

The logic **SIL** also allows direct comparisons between indices and values. For instance, the formula $\forall i \,.\, a[i] = i + 1$ is satisfied by all arrays a which are of the form $1234\ldots$. Alternatively, this can be specified as $a[0] = 1 \;\wedge\; \forall i \,.\, a[i+1] = a[i] + 1$ where $a[0] = 1$ is a shorthand for $\forall i \,.\, i = 0 \rightarrow a[i] = 1$. Further, the set of arrays in which the value at position n is between zero and n can be specified by writing $\forall i \,.\, 0 \leq a[i] < i$, which cannot be described without an explicit comparison between indices and values (unless a comparison with an additional array describing the sequence $1234\ldots$ is used).

Checking verification conditions for array manipulating programs. The decision procedure for checking satisfiability of **SIL** formulae, described later on, can be used for discharging verification conditions of various interesting array-manipulating procedures. As a concrete example, let us consider the procedure for an in-situ left rotation of arrays, given below. We annotate the procedure (using double braces) with a pre-condition, post-condition, and a loop invariant. We distinguish below logical variables from program variables (typeset in print). The variable a_0 is a logical variable that relates the initial values of the array `a` with the values after the rotation.

$\{\{\ |\mathtt{a}| = |a_0| \wedge \forall j.\mathtt{a}[j] = a_0[j]\ \}\}$
`x=a[0];`
for `(i=0; i <` $|\mathtt{a}|$`-1; i++)`
  $\{\{\ \mathtt{x} = a_0[0] \wedge \forall j.0 \leq j < \mathtt{i} \rightarrow \mathtt{a}[j] = a_0[j+1] \wedge \forall j.\mathtt{i} \leq j < |\mathtt{a}| \rightarrow \mathtt{a}[j] = a_0[j]\ \}\}$
    `a[i]=a[i+1];`
`a[`$|\mathtt{a}|$`-1]=x;`
$\{\{\ \mathtt{a}[|\mathtt{a}| - 1] = a_0[0] \wedge \forall j.0 \leq j < |\mathtt{a}| - 1 \rightarrow \mathtt{a}[j] = a_0[j+1])\ \}\}$

To check (partial) correctness of the procedure, one needs to check three verification conditions out of which we discuss one here (the others are similar). Namely, we consider checking the loop invariant, which requires checking validity of the formula:

$$\begin{array}{c} \mathtt{x} = a_0[0] \wedge \forall j.0 \leq j < \mathtt{i} \rightarrow \mathtt{a}[j] = a_0[j+1] \ \wedge \ \forall j.i \leq j < |\mathtt{a}| \rightarrow \mathtt{a}[j] = a_0[j] \ \wedge \\ \mathtt{i} < |\mathtt{a}| - 1 \wedge |\mathtt{a}'| = |\mathtt{a}| \wedge \mathtt{i}' = \mathtt{i}+1 \wedge \mathtt{x}' = \mathtt{x} \wedge \mathtt{a}'[\mathtt{i}] = \mathtt{a}[\mathtt{i}+1] \wedge \forall j.j \neq \mathtt{i} \rightarrow \mathtt{a}'[j] = \mathtt{a}[j] \\ \longrightarrow \\ \mathtt{x}' = a_0[0] \wedge \forall j.0 \leq j < \mathtt{i}' \rightarrow \mathtt{a}'[j] = a_0[j+1] \ \wedge \ \forall j.i' \leq j < |\mathtt{a}'| \rightarrow \mathtt{a}'[j] = a_0[j] \end{array}$$

Primed variables denote the values of program variables after one iteration of the loop. Checking validity of this formula amounts to checking that its negation is unsatisfiable. The latter condition is expressible in the decidable fragment of **SIL**. Note that the conditions used above refer to adjacent array positions, which could not be expressed in the logics defined in [2,5,3].

4 Undecidability of the Logic SIL

In this section, we show that the satisfiability problem for the $\forall^*\exists^*\forall^*$ fragment of **SIL** is undecidable, by reducing from Hilbert's Tenth Problem [14]. In the following, Section 5 proves the decidability of the satisfiability problem for the fragment of boolean combinations of universally quantified array property formulae—the satisfiability of the $\forall^*$ fragment is proven. Since the leading existential prefix is irrelevant when one speaks about satisfiability, referring either to $\forall^*\exists^*\forall^*$ or to $\exists^*\forall^*\exists^*\forall^*$ makes no difference in this case. However, the question concerning the *validity* problem for the $\exists^*\forall^*$ fragment of **SIL** is still open.

First, we show that multiplication and addition of strictly positive integers can be encoded using formulae of $\forall^*\exists^*\forall^*$-**SIL**. Let $x, y, z \in \mathbb{N}$, with $z > 0$. We define:

$$\begin{array}{l} \varphi_1(j) : a_2[j] > 0 \wedge a_3[j] > 0 \wedge a_1[j+1] = a_1[j] + 1 \wedge a_2[j+1] = a_2[j] - 1 \wedge \\ \qquad \wedge \ a_3[j+1] = a_3[j] \end{array}$$

$$\varphi_2(j) : a_2[j] = 0 \wedge a_3[j] > 0 \wedge a_1[j+1] = a_1[j] \wedge a_2[j+1] = y \wedge a_3[j+1] = a_3[j] - 1$$

$$\begin{array}{l} \varphi_{x=yz}(a_1, a_2, a_3, n_1, n_2) : n_1 < n_2 \wedge a_1[n_1] = 0 \wedge a_2[n_1] = y \wedge a_3[n_1] = z \wedge a_1[n_2] = x \wedge \\ \qquad \wedge \ a_3[n_2] = 0 \wedge \forall i.(n_1 \leq i < n_2 \rightarrow \exists j.i \leq j < n_2 \wedge \varphi_2(j) \wedge \forall k.(i \leq k < j \rightarrow \varphi_1(k))) \end{array}$$

Notice that $\varphi_{x=yz}$ is in the $\forall^*\exists^*\forall^*$ quantifier fragment of **SIL**.

Lemma 3. *$\varphi_{x=yz}(a_1, a_2, a_3, n_1, n_2)$ is satisfiable if and only if $x = yz$.*

Proof. We first suppose that $x = yz$ and give a model of $\varphi_{x=yz}(a_1, a_2, a_3, n_1, n_2)$. We choose $n_1 = 0$ and $n_2 = (y+1)z$. Then, we choose $a_1[n_2] = x$, $a_2[n_2] = y$ and $a_3[n_2] = 0$. Furthermore, for all j such that $0 \leq j < z$ and for all i such that $0 \leq i \leq y$, we choose $a_1[i + j(y+1)] = i + jy$, $a_2[i + j(y+1)] = y - i$ and $a_3[i + j(y+1)] = z - j$. Then, it is easy to check that this is a model of $\varphi_{x=yz}(a_1, a_2, a_3, n_1, n_2)$.

Let us consider now a model of $\varphi_{x=yz}(a_1,a_2,a_3,n_1,n_2)$. We show that this implies $x=yz$. A model of $n_1<n_2\wedge a_1[n_1]=0\wedge a_2[n_1]=y\wedge a_3[n_1]=z\wedge a_1[n_2]=x\wedge a_3[n_2]=0\wedge\forall i.(n_1\le i<n_2\rightarrow\exists j.i\le j<n_2\wedge\varphi_2(j)\wedge\forall k.(i\le k<j\rightarrow\varphi_1(k)))$ assigns values to n_1 and n_2 and defines array values for a_1, a_2, and a_3 between bounds n_1 and n_2. Clearly, $a_1[n_1]=0$, $a_2[n_1]=y$, $a_3[n_1]=z$, $a_1[n_2]=x$, and $a_3[n_2]=0$. Due to their definition, $\varphi_1(j)$ and $\varphi_2(j)$ cannot be true at the same point j since $\models\varphi_1(j)\rightarrow a_2[j]>0$ and $\models\varphi_2(j)\rightarrow a_2[j]=0$.

Since the subformula $\forall i.(n_1\le i<n_2\rightarrow\exists j.i\le j<n_2\wedge\varphi_2(j)\wedge\forall k.(i\le k<j\rightarrow\varphi_1(k)))$ holds, it is then clear that there exists points $j_1,\ldots,j_l$ with $l>0$ and $n_1\le j_1<j_2<\cdots<j_l=n_2-1$ such that $\varphi_2(j)$ holds at all of these points. Furthermore, at all intermediary points k not equal to one of the j_i's, $\varphi_1(k)$ has to be true. This implies that l must be equal to z (since $\varphi_1(k)$ imposes $a_3[k+1]=a_3[k]$ whereas $\varphi_2(j)$ imposes $a_3[j+1]=a_3[j]-1$).

Let us examine the intermediary points between n_1 and j_1. Due to $a_1[n_1]=0$, $a_2[n_1]=y$, $a_3[n_1]=z$ and $\varphi_1(k)$ being true for all k such that $n_1\le k<j_1$ as well as $\varphi_2(j_1)$ being true, we must have $j_1=y+n_1$, and, for all k such that $n_1<k\le j_1$, we have $a_1[k]=k-n_1$, $a_2[k]=y-k+n_1$, and $a_3[k]=z$. Furthermore, since $\varphi_2(j_1)$ is true, we have $a_1[j_1+1]=y$, $a_2[j_1+1]=y$, and $a_3[j_1+1]=z-1$. We can continue this reasoning with the intermediary points between j_1 and j_2 and so on up to j_l. At the end we get $a_3[j_l+1]=0$ and $a_1[j_l+1]=a_1[n_2]=yl$. Since $l=z$ and $a_1[n_2]=x$, this implies $x=yz$. □

Next, we define:

$$\varphi_3(j):a_2[j]>0\wedge a_1[j+1]=a_1[j]+1\wedge a_2[j+1]=a_2[j]-1$$

$$\varphi_{x=y+z}(a_1,a_2,n_1,n_2):n_1<n_2\wedge a_1[n_1]=y\wedge a_2[n_1]=z\wedge a_1[n_2]=x\wedge a_2[n_2]=0\ \wedge\\ \wedge\ \forall k.n_1\le k<n_2\rightarrow\varphi_3(k)$$

Lemma 4. *$\varphi_{x=y+z}(a_1,a_2,n_1,n_2)$ is satisfiable if and only if $x=y+z$.*

Proof. Similar to Lemma 3. □

We are now ready to reduce from Hilbert's Tenth Problem [14]. Given a Diophantine system S, we construct a **SIL** formula Ψ_S which is satisfiable if and only if the system has a solution. Without loss of generality, we can suppose that all variables in S range over strictly positive integers. Then S can be equivalently written as a system of equations of the form $x=yz$ and $x=y+z$ by introducing fresh variables. Let $\{x_1,\ldots,x_k\}$ be the variables of these equations. We enumerate separately all equations of the form $x=yz$ and those of the form $x=y+z$. Let n_m be the number of equations of the form $x=yz$ and n_a the number of equations of the form $x=y+z$.

Let Ψ_S be the following **SIL** formula with three array symbols (a_1,a_2 and a_3):

$$\exists x_1\ldots\exists x_k\exists m_1^1\ldots\exists m_{n_m+n_a}^1\exists m_1^2\ldots\exists m_{n_m+n_a}^2\bigwedge_{i=1}^{n_m+n_a-1}m_i^2<m_{i+1}^1\wedge\bigwedge_{i=1}^{n_m}\varphi_i\wedge\bigwedge_{i=1}^{n_a}\varphi_i'$$

where the formulae φ_i and φ_i' are defined as follows: Let $x_{i_1}=x_{i_2}x_{i_3}$ be the i-th multiplicative equation. Then, $\varphi_i=\varphi_{x_{i_1}=x_{i_2}x_{i_3}}(a_1,a_2,a_3,m_i^1,m_i^2)$. Let $x_{i_1}=x_{i_2}+x_{i_3}$ be the i-th additive equation. Then, $\varphi_i'=\varphi_{x_{i_1}=x_{i_2}+x_{i_3}}(a_1,a_2,m_{n_m+i}^1,m_{n_m+i}^2)$.

Lemma 5. *A Diophantine system S has a solution if and only if the corresponding formula Ψ_S is satisfiable.*

Proof. The Diophantine system S is equivalently written as a conjunction of equations of the form $x = yz$ and $x = y + z$ using variables $\{x_1, \ldots, x_k\}$. Then, the Diophantine system has a solution if and only if all equations of the form $x = yz$ and $x = y + z$ have a common solution. Since all pairs m_i^1 and m_i^2 denote disjoint intervals and using Lemmas 3 and 4, we have that all equations of the form $x = yz$ and $x = y + z$ have a common solution if and only if Ψ_S is satisfiable. □

5 Decidability of the Satisfiability Problem for $\exists^*\forall^*$-SIL

We show that the set of models of a boolean combination φ of universally quantified array property formulae of **SIL** corresponds to the set of runs of an FCADBM A_φ, defined inductively on the structure of the formula. More precisely, each array variable in φ has a corresponding counter in A_φ, and given any model of φ that associates integer values to all array entries, A_φ has a run in which the values of the counters at different points of the run match the values of the array entries at corresponding positions in the model. Since the emptiness problem is decidable for FCADBM, this leads to decidability of the satisfiability problem for $\exists^*\forall^*$-**SIL** (or equivalently, for $\forall^*$-**SIL**).

5.1 Normalisation

Before describing the translation of $\exists^*\forall^*$-**SIL** formulae into counter automata, we need to perform a simple normalisation step. Let $\varphi(\mathbf{k}, \mathbf{a})$ be a **SIL** formula in the $\exists^*\forall^*$ fragment i.e., an existentially quantified boolean combination of (1) DBM conditions or modulo constraints on array-bound variables $\mathbf{k}$ and array length terms $|a|$, $a \in \mathbf{a}$, and (2) array properties of the form $\forall i \,.\, \gamma(i, \mathbf{k}, |\mathbf{a}|) \rightarrow \upsilon(i, \mathbf{k}, \mathbf{a})$[4]. Without losing generality, we assume that the sanity condition $\mathcal{B}(\upsilon)$ is explicitly conjoined to the guard of every array property i.e., each array property is of the form $\forall i \,.\, \gamma \wedge \mathcal{B}(\upsilon) \rightarrow \upsilon$.

A *guard expression* is a conjunction of array-bound expressions $i \sim \ell$, $\sim \in \{\leq, \geq\}$, or modulo constraints $i \equiv_s t$ where ℓ is a an array bound term, and $s, t \in \mathbb{N}$ such that $0 \leq t < s$. For a guard γ and an integer constant $c \in \mathbb{Z}$, we denote by $\gamma + c$ the guard obtained by replacing each array-bound expression $i \sim b$ by $i \sim b + c$ and each modulo constraint $i \equiv_s t$ by $i \equiv_s t'$ where $0 \leq t' < s$ and $t' \equiv_s t + c$.

The normalisation consists in performing the following steps in succession:

1. Replace each array property subformula $\forall i \,.\, \bigvee_j \gamma_j \rightarrow \bigwedge_k \upsilon_k$ by the equivalent conjunction $\bigwedge_{j,k} \forall i \,.\, \gamma_j \rightarrow \upsilon_k$ where γ_j are guard expressions and υ_k are either $a[i+n] \sim \ell$, $a[i+n] - b[i+m] \sim p$, or $i - a[i+n] \sim m$, where $m, n, p \in \mathbb{Z}$, $\sim \in \{\leq, \geq\}$ and ℓ is an array bound term.

[4] An array property formula with more than one universally quantified index variable in the guard is equivalent to an array property formula whose guard has exactly one universally quantified index variable. Indeed, a formula of the form $\forall i_1, \ldots, i_n \,.\, \gamma(i_1, \ldots, i_n, \mathbf{k}, |\mathbf{a}|) \rightarrow \upsilon(i_1, \mathbf{k}, \mathbf{a})$ is equivalent to $\forall i_1 \,.\, ((\exists i_2, \ldots, i_n \,.\, \gamma(i_1, \ldots, i_n, \mathbf{k}, |\mathbf{a}|)) \rightarrow \upsilon(i_1, \mathbf{k}, \mathbf{a}))$ and then the existential quantifiers in $(\exists i_2, \ldots, i_n \,.\, \gamma(i_1, \ldots, i_n, \mathbf{k}, |\mathbf{a}|))$ can be eliminated possibly adding modulo constraints on $\mathbf{k}$, $|\mathbf{a}|$ and i_1.

2. Simplify each newly obtained array property subformula as follows:

$$\begin{aligned}
\forall i \,.\, \gamma \to a[i+n] \sim \ell &\;\rightsquigarrow\; \forall i \,.\, \gamma + n \to a[i] \sim \ell \\
\forall i \,.\, \gamma \to i - a[i+n] \sim m &\;\rightsquigarrow\; \forall i \,.\, \gamma + n \to i - a[i] \sim m + n \\
\forall i \,.\, \gamma \to a[i+n] - b[i+m] \sim p &\;\rightsquigarrow\; \forall i \,.\, \gamma + n \to a[i] - b[i+m-n] \sim p \text{ if } m \geq n \\
\forall i \,.\, \gamma \to a[i+n] - b[i+m] \sim p &\;\rightsquigarrow\; \forall i \,.\, \gamma + m \to b[i] - a[i+n-m] \overline{\sim} - p \text{ if } m < n
\end{aligned}$$

   where:
   - $\sim \in \{\leq, \geq\}$ and $\overline{\sim}$ is $\geq$ ($\leq$) if $\sim$ is $\leq$ ($\geq$), respectively, and
   - ℓ is an array-bound term, and $m, n, p \in \mathbb{Z}$.
3. For each array property $\psi : \forall i \,.\, \gamma(i) \to \upsilon(i)$, let $B_\psi = \{b_1, \ldots, b_n\}$ be the set of array-bound terms occurring in γ. Then replace ψ by the disjunction $\bigvee_{1 \leq i,j \leq n} \bigwedge_{1 \leq k \leq n} b_i \leq b_k \leq b_j \wedge \psi$ (one considers all possible cases of minimal and maximal values for array-bound terms), and simplify all subformulae of the form $\bigwedge_j i \leq b_j$ ($\bigwedge_j i \geq b_j$) from γ to exactly one upper (lower) bound, according to the current conjunctive clause. If the lower and upper bound that appear in γ are inconsistent with the chosen minimal and maximal value added by the transformation to ψ (i.e., the lower bound is assumed to be bigger than the upper one), we replace ψ in the concerned conjunctive clause by $\top$ as it is trivially satisfied.
4. Rewrite each conjunction $\bigwedge_j i \equiv_{s_j} t_j$ occurring within the guards of array property formulae into $\bigwedge_j i \equiv_S \frac{S \cdot t_j}{s_j}$ where S is the least common multiple of s_j, and simplify the conjunction either to false (in which case the array property subformula is vacuously true), or to a formula $i \equiv_S t$. In case there is no modulo constraint within a guard, for uniformity reasons, conjoin the guard with the constraint $i \equiv_1 0$.
5. Transform each array property subformula of the form

$$\forall i \,.\, f \leq i \leq g \wedge i \equiv_s t \;\longrightarrow\; a[i] - b[i+m] \sim n$$

   where $m > 1$, $n \in \mathbb{Z}$, and $0 \leq t < s$ into the following conjunction:

$$\begin{aligned}
&\forall i \,.\, f \leq i \leq g \wedge i \equiv_s t \;\longrightarrow\; a[i] - \tau_1[i+1] \sim 0 \;\wedge \\
&\textstyle\bigwedge_{j=1}^{m-2} \forall i . f + j \leq i \leq g + j \wedge i \equiv_s (t+j) \bmod s \;\longrightarrow\; \tau_j[i] - \tau_{j+1}[i+1] \sim 0 \;\wedge \\
&\forall i \,.\, f + m - 1 \leq i \leq g + m - 1 \wedge i \equiv_s (t+m-1) \bmod s \;\longrightarrow\; \tau_{m-1}[i] - b[i+1] \sim n
\end{aligned}$$

   where $\tau_1, \tau_2, \ldots, \tau_{m-1}$ are fresh array variables. Figure 2 depicts this transformation for $\sim = \leq$ – the case $\sim = \geq$ is similar.

The result of the normalisation step is a boolean combination of (1) DBM conditions or modulo constraints on array-bound variables $\mathbf{k}$ and array length terms $|a|$, $a \in \mathbf{a}$ and (2) array properties of the following form:

$$\forall i \,.\, f \leq i \leq g \;\wedge\; i \equiv_s t \to \upsilon$$

where f and g are array-bound terms, $s, t \in \mathbb{N}$, $0 \leq s < t$, and υ is one of the following:

$$(1)\; a[i] \sim \ell, \quad (2)\; i - a[i] \sim n, \quad (3)\; a[i] - b[i+1] \sim n$$

where $\sim \in \{\leq, \geq\}$, $n \in \mathbb{Z}$, and ℓ is an array-bound term.

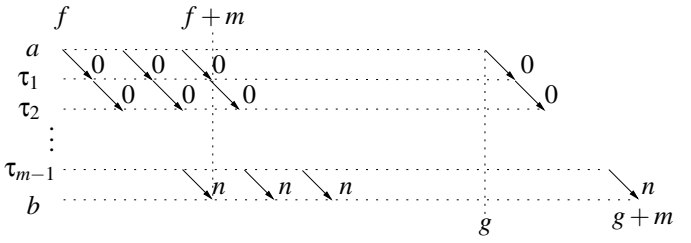


Fig. 2. Adding fresh array variables to array property formulae $\forall i \,.\, f \leq i \wedge i \leq g \wedge i \equiv_s t \rightarrow a[i] - b[i+m] \leq n$

We need the following definition to state the normal form lemma. If $X \subseteq AVar$ is a set of array variables, then $\mu{\downarrow_X}$ represents the restriction of $\mu : AVar \rightarrow \mathbb{Z}^*$ to the variables in X. For a formula φ of **SIL**, we denote by $[\![\varphi]\!]{\downarrow_X}$ the set $\{\langle \iota, \mu{\downarrow_X}\rangle \mid \langle \iota, \mu\rangle \models \varphi\}$.

Lemma 6. *Let* $\varphi(\mathbf{k}, \mathbf{a})$ *be a formula of* $\exists^*\forall^*$**-SIL** *and* $\phi(\mathbf{k}, \mathbf{a}, \mathbf{t})$ *be the formula obtained from* φ *by normalisation where* $\mathbf{t}$ *is the set of fresh array variables added during normalisation. Then we have* $[\![\varphi]\!] = [\![\phi]\!]{\downarrow_{\mathbf{a}}}$.

5.2 Translating Normalised Formulae into FCADBM

Let $\varphi(\mathbf{k}, \mathbf{a})$ be an $\exists^*\forall^*$**-SIL** formula that is already normalised as in the previous. The automaton encoding the models of φ is in fact a product $\mathcal{A}_\varphi = A_\varphi \otimes A_{tick}$, where A_φ is defined inductively on the structure of φ, and A_{tick} is a generic FCADBM, defined next. Both A_φ and A_{tick} (and, implicitly $\mathcal{A}_\varphi$) work with the set of counters $\mathbf{x} = \{x_k \mid k \in \mathbf{k}\} \cup \{x_{|a|} \mid a \in \mathbf{a}\} \cup \{x_a \mid a \in \mathbf{a}\} \cup \{x_{tick}\}$, where:

- x_k and $x_{|a|}$ are parameters corresponding to array-bound variables, i.e., their values do not change during the runs of $\mathcal{A}_\varphi$,
- x_a are counters corresponding to the array symbols, and
- x_{tick} is a special counter that is initialised to zero and incremented by each transition.

The main intuition behind the automata construction is that, for each model $\langle \iota, \mu\rangle$ of φ, there exists a run of $\mathcal{A}_\varphi$ such that, for each array symbol $a \in \mathbf{a}$, the value $\mu(a)_n$ equals the value of x_a when x_{tick} equals n, for all $0 \leq n < |a|$. The reason behind defining $\mathcal{A}_\varphi$ as the product of A_φ and A_{tick} is that the use of negation within φ, which involves complementation on the automata level, may not affect the flow of ticks, just the way they are dealt with within the guards. For this reason, A_φ can only read x_τ, while A_{tick} is the one updating it.

Formally, let $A_{tick} = \langle \mathbf{x}, \{q_0, q_{tick}\}, \{q_0\}, \rightarrow_{tick}, \{q_{tick}\}\rangle$, where

$$q_0 \xrightarrow{x_{tick}=0 \,\wedge\, x'_{tick}=x_{tick}+1 \,\wedge\, \bigwedge_{k\in\mathbf{k}} x'_k=x_k \,\wedge\, \bigwedge_{a\in\mathbf{a}} x'_{|a|}=x_{|a|}} q_{tick}$$

and

$$q_{tick} \xrightarrow{x'_{tick}=x_{tick}+1 \,\wedge\, \bigwedge_{k\in\mathbf{k}} x'_k=x_k \,\wedge\, \bigwedge_{a\in\mathbf{a}} x'_{|a|}=x_{|a|}} q_{tick}$$

are the only transitions rules. The construction of A_φ is recursive on the structure of φ:

- if φ is a DBM constraint or modulo constraint θ on array-bound terms, let $A_\varphi = \langle \mathbf{x}, \{q_0, q_1\}, \{q_0\}, \rightarrow, \{q_1\} \rangle$ where the transitions rules are $q_1 \xrightarrow{\top} q_1$ and $q_0 \xrightarrow{\overline{\theta}} q_1$, and $\overline{\theta}$ is obtained from the constraint θ by replacing all occurrences of $k \in \mathbf{k}$ by x_k, and all occurrences of $|a|$, $a \in \mathbf{a}$, by $x_{|a|}$.
- if $\varphi = \neg\psi$, let $A_\varphi = \overline{A_\psi}$,
- if $\varphi = \psi_1 \wedge \psi_2$, let $A_\varphi = A_{\psi_1} \otimes A_{\psi_2}$,
- if $\varphi = \psi_1 \vee \psi_2$, let $A_\varphi = \overline{\overline{A_{\psi_1}} \otimes \overline{A_{\psi_2}}}$.
- if φ is an array property, A_φ is defined below, according to the type of the value expression occurring on the right hand side of the implication.

Let $\varphi : \forall i \,.\, f \leq i \leq g \;\wedge\; i \equiv_s t \rightarrow \upsilon$ be an array property subformula after normalisation. Figure 3 gives the counter automaton A_φ for such a subformula. The formal definition of $A_\varphi = \langle \mathbf{x}, Q, I, \rightarrow, F \rangle$ follows:

- $Q = \{q_i \mid 0 \leq i < s\} \cup \{r_i \mid 0 \leq i < s\} \cup \{q_f\}$, $I = \{q_0\}$, and $F = \{q_f\}$.
- the transition rules of A_φ are as follows, for all $0 \leq i < s$:

$q_i \xrightarrow{x_{tick} < \overline{f} - 1} q_{(i+1) \bmod s}$ $\quad$ $q_i \xrightarrow{x_{tick} = \overline{f} - 1} r_{(i+1) \bmod s}$

$r_i \xrightarrow{\overline{f} \leq x_{tick} \leq \overline{g}} r_{(i+1) \bmod s}$ if $i \neq t$ $\quad$ $r_t \xrightarrow{\overline{\upsilon} \wedge \overline{f} \leq x_{tick} \leq \overline{g}} r_{(t+1) \bmod s}$

$r_i \xrightarrow{x_{tick} > \overline{g}} q_f$ $\quad$ $q_f \xrightarrow{\top} q_f$

$q_0 \xrightarrow{x_{tick} = 0 \wedge \overline{\upsilon} \wedge \overline{f} \leq x_{tick} \leq \overline{g}} r_{1 \bmod s}$ if $t = 0$ $\quad$ $q_0 \xrightarrow{x_{tick} = 0 \wedge \overline{f} \leq x_{tick} \leq \overline{g}} r_{1 \bmod s}$ if $t \neq 0$

Here $\overline{\upsilon}$ is defined by:

- $\overline{\upsilon} \triangleq x_a \sim \overline{\ell}$ if υ is $a[i] \sim \ell$ where $\overline{\ell}$ is obtained from ℓ by replacing each occurrence of $k \in \mathbf{k}$ by x_k and each occurrence of $|a|$ by $x_{|a|}$, $a \in \mathbf{a}$,
- $\overline{\upsilon} \triangleq x_{tick} - x_a \sim n$ if υ is $i - a[i] \sim n$, and
- $\overline{\upsilon} \triangleq x_a - x'_b \sim n$ if υ is $a[i] - b[i+1] \sim n$.

Further, $\overline{f}$ ($\overline{g}$) are obtained from f (g) by replacing each $k \in \mathbf{k}$ by x_k and each $|a|$, $a \in \mathbf{a}$, by $x_{|a|}$, respectively.

Notice that A_φ is always deterministic. This is because the automata for array property formulae are deterministic in the use of the x_{tick} counter, complementation preserves determinism, and composition of two deterministic FCADBM results in a deterministic FCADBM.

Let $\varphi(\mathbf{k}, \mathbf{a})$ be a normalised $\exists^*\forall^*$-**SIL** formula, and $\mathcal{A}_\varphi = A_\varphi \otimes A_{tick}$ be the deterministic FCADBM whose construction was given in the previous. We define the following relation between valuations $\langle \iota, \mu \rangle \in [\![\varphi]\!]$ and traces $\sigma \in Tr(\mathcal{A}_\varphi)$, denoted $\langle \iota, \mu \rangle \equiv \sigma$, iff:

1. for all $k \in \mathbf{k}$, $\iota(k) = \sigma_0(x_k)$,
2. for all $a \in \mathbf{a}$, $\iota(|a|) = \sigma_0(x_{|a|}) = |\mu(a)| \leq |\sigma|$ and $\mu(a)_i = \sigma_i(x_a)$, $0 \leq i < |\mu(a)|$.

The following lemma establishes correctness of our construction:

Lemma 7. *Let $\varphi(\mathbf{k}, \mathbf{a})$ be a normalised $\exists^*\forall^*$-**SIL** formula, and $\mathcal{A}_\varphi$ be its corresponding FCADBM. Then for each valuation $\langle \iota, \mu \rangle \in [\![\varphi]\!]$ there exist a trace $\sigma \in Tr(\mathcal{A}_\varphi)$ such that $\langle \iota, \mu \rangle \equiv \sigma$. Dually, for each trace $\sigma \in Tr(\mathcal{A}_\varphi)$ there exists a valuation $\langle \iota, \mu \rangle \in [\![\varphi]\!]$ such that $\langle \iota, \mu \rangle \equiv \sigma$.*

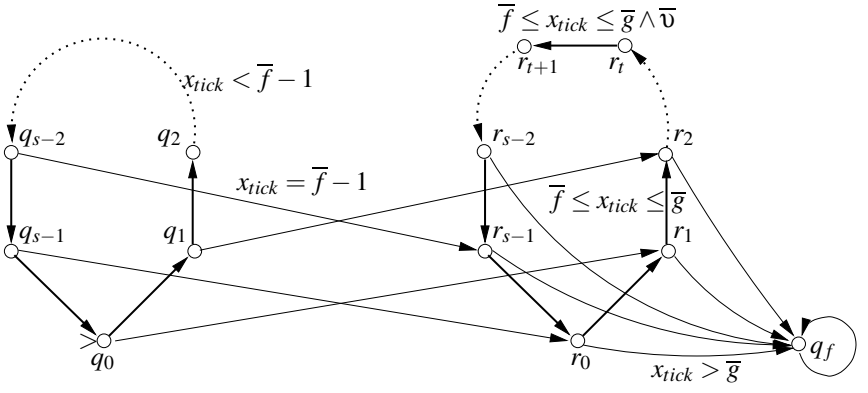


Fig. 3. The FCADBM for the formula $\forall i \,.\, f \leq i \leq g \;\wedge\; i \equiv_s t \rightarrow \upsilon$

Theorem 2. *The satisfiability problem is decidable for the* $\exists^*\forall^*$ *fragment of* **SIL**.

Proof. Let $\varphi(\mathbf{k},\mathbf{a})$ be a formula of $\exists^*\forall^*$-**SIL**. By normalisation, we obtain a formula $\phi(\mathbf{k},\mathbf{a},\mathbf{t})$ where $\mathbf{t}$ is the set of fresh array variables added during normalisation. Then, by Lemma 6, we have $[\![\varphi]\!] = [\![\phi]\!]\downarrow_{\mathbf{a}}$. To check satisfiability of φ, it is therefore enough to check satisfiability of ϕ. By Lemma 7, ϕ is satisfiable if and only if the language of the corresponding automaton $\mathcal{A}_\phi$ is not empty. This is decidable by Theorem 1. □

6 Conclusion

We have introduced a logic over integer arrays based on universally quantified difference bound constraints on array elements situated within a constant sized window. We have shown that the logic is undecidable for formulae with quantifier prefix in the language $\forall^*\exists^*\forall^*$, and that the $\exists^*\forall^*$ fragment is decidable. This is shown with an automata-theoretic argument by constructing, for a given formula, a corresponding equivalent counter automaton whose emptiness problem is decidable. The translation of formulae into counter automata takes advantage of the fact that only one index is used in the difference bound constraints on array values, making the decision procedure for the logic simple and efficient. Future work involves automatic invariant generation for programs handling arrays, as well as implementation and experimental evaluation of the method.

References

1. Armando, A., Ranise, S., Rusinowitch, M.: Uniform derivation of decision procedures by superposition. In: Fribourg, L. (ed.) CSL 2001 and EACSL 2001. LNCS, vol. 2142, p. 513. Springer, Heidelberg (2001)
2. Arons, T., Pnueli, A., Ruah, S., Xu, J., Zuck, L.: Parameterized verification with automatically computed inductive assertions. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102. Springer, Heidelberg (2001)

3. Bouajjani, A., Jurski, Y., Sighireanu, M.: A generic framework for reasoning about dynamic networks of infinite-state processes. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424. Springer, Heidelberg (2007)
4. Bozga, M., Iosif, R., Lakhnech, Y.: Flat parametric counter automata. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) ICALP 2006. LNCS, vol. 4052. Springer, Heidelberg (2006)
5. Bradley, A.R., Manna, Z., Sipma, H.B.: What's decidable about arrays? In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855. Springer, Heidelberg (2006)
6. Comon, H., Jurski, Y.: Multiple Counters Automata, Safety Analysis and Presburger Arithmetic. In: Vardi, M.Y. (ed.) CAV 1998. LNCS, vol. 1427. Springer, Heidelberg (1998)
7. The FLATA Toolset, http://www-verimag.imag.fr/~async/FLATA/flata.html
8. Ghilardi, S., Nicolini, E., Ranise, S., Zucchelli, D.: Decision Procedures for Extensions of the Theory of Arrays. Annals of Mathematics and Artificial Intelligence, 50 (2007)
9. Habermehl, P., Iosif, R., Vojnar, T.: A Logic of Singly Indexed Arrays. Technical Report TR-2008-9, Verimag (2008)
10. Habermehl, P., Iosif, R., Vojnar, T.: What else is decidable about integer arrays? In: Amadio, R. (ed.) FOSSACS 2008. LNCS, vol. 4962. Springer, Heidelberg (2008)
11. Jaffar, J.: Presburger Arithmetic with Array Segments. Inform. Processing Letters, 12 (1981)
12. King, J.: A Program Verifier. PhD thesis, Carnegie Mellon University (1969)
13. Mateti, P.: A Decision Proc. for the Correctness of a Class of Programs. Journal of the ACM, 28(2) (1980)
14. Matiyasevich, Y.: Enumerable Sets are Diophantine. Journal of Sovietic Mathematics 11, 354–358 (1970)
15. McCarthy, J.: Towards a Mathematical Science of Computation. In: IFIP Congress (1962)
16. Stump, A., Barrett, C.W., Dill, D.L., Levitt, J.R.: A Decision Procedure for an Extensional Theory of Arrays. In: Proc. of LICS 2001 (2001)
17. Suzuki, N., Jefferson, D.: Verification Decidability of Presburger Array Programs. Journal of the ACM 27(1) (1980)

On the Computational Complexity of Spatial Logics with Connectedness Constraints

R. Kontchakov[1], I. Pratt-Hartmann[2], F. Wolter[3], and M. Zakharyaschev[1]

[1] School of Computer Science, Birkbeck College London
[2] Department of Computer Science, Manchester University
[3] Department of Computer Science, University of Liverpool

Abstract. We investigate the computational complexity of spatial logics extended with the means to represent topological connectedness and restrict the number of connected components. In particular, we show that the connectedness constraints can increase complexity from NP to PSpace, ExpTime and, if component counting is allowed, to NExpTime.

1 Introduction

A subset of a topological space T is *connected* if it cannot be covered by the union of two disjoint non-empty open sets in T. Connectedness is known to be one of the most fundamental concepts of topology, and any textbook in the field contains a substantial chapter on connectedness. In spatial representation and reasoning in AI, the distinction between connected and disconnected regions is recognized as indispensable for various modelling and representation tasks; see, e.g., [1,4]. (After all, a disconnected plot is usually only worth half the value of a connected plot.) In spite of this, so far only sporadic attempts have been made to investigate the computational complexity of spatial logics with connectedness constraints [3,21,23,15].

In this paper, we consider extensions of standard spatial logics designed for qualitative spatial representation and reasoning (see, e.g., [18,4] for recent surveys) with connectedness constraints such as 'region r is connected' (or $c(r)$, in symbols) and 'region r contains at most k connected components' (or $c^{\leq k}(r)$). Our main aim is to provide a systematic study of the impact of these constraints on the computational complexity of the satisfiability problem. We focus only on quantifier-free spatial logics because first-order qualitative theories of topological spaces are generally undecidable or non-recursively enumerable even without connectedness constraints [10,7,5,12].

The weakest spatial formalisms for which the addition of connectedness constraints is of interest appear to be '9-intersections' and $\mathcal{RCC}$-8 [8,16], where one can relate regions (regular closed sets) using binary predicates such as mereological $\mathsf{O}(r, s)$ ('regions r and s overlap') or mereotopological $\mathsf{EC}(r, s)$ ('regions r and s are externally connected'). However, as far as satisfiability is concerned, these logics cannot distinguish between arbitrary regions, connected regions, or regions with k connected components [17], primarily because no Boolean operators on

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 574–589, 2008.

regions are available in their languages. That is why the weakest spatial formalism, $\mathcal{B}$, considered in this paper consists of only Boolean region terms denoting Boolean combinations of regions. $\mathcal{B}$ itself is also rather weak (in fact, reasoning in $\mathcal{B}$ coincides with Boolean reasoning about sets), but we show that its extensions $\mathcal{B}c$ and $\mathcal{B}cc$ with constraints $c(r)$ and $c^{\leq k}(r)$, respectively, are full-fledged topological logics with considerably more expressive power. Moreover—and this was quite an unexpected result for the authors—the computational complexity jumps from NP for $\mathcal{B}$ to ExpTime for $\mathcal{B}c$ and NExpTime for $\mathcal{B}cc$.

Another spatial logic we deal with in this paper is $\mathcal{BRCC}$-8 [23] which extends $\mathcal{RCC}$-8 with Boolean region terms. An equivalent formalism was also considered in the framework of Boolean contact algebras by extending the Boolean algebra of regular closed (or open) sets with Whitehead's 'extensive connection' predicate $C(r, s)$; see [22,6]. Here we denote this logic by $\mathcal{C}$ (in order to unify the two lines of research). As shown in [23], $\mathcal{C}$ is still NP-complete. We prove, however, that its extensions $\mathcal{C}c$ and $\mathcal{C}cc$ with constraints of the form $c(r)$ and $c^{\leq k}(r)$ are also ExpTime-complete and NExpTime-complete, respectively. Our maximal spatial logic has its roots in the seminal paper by McKinsey and Tarski [13]. Following the modal logic tradition, we call it $\mathcal{S}4_u$ ($\mathcal{S}4$ with the universal modality). In contrast to $\mathcal{B}$ and $\mathcal{C}$, $\mathcal{S}4_u$ is PSpace-complete. Its extensions $\mathcal{S}4_u c$ and $\mathcal{S}4_u cc$, however, turn out to be ExpTime-complete and NExpTime-complete again.

Thus, the addition of connectedness constraints to standard spatial logics with Boolean region terms leads to considerably more expressive languages of higher computational complexity. However, this increase in complexity is 'stable:' the extensions $\mathcal{B}c$ and $\mathcal{S}4_u c$ of such different formalisms as $\mathcal{B}$ and $\mathcal{S}4_u$ are of the same complexity. Another interesting result is that by restricting these languages to formulas with just one connectedness constraint of the form $c(r)$, we obtain logics that are still in PSpace, but two such constraints lead to ExpTime-hardness. In fact, if the connectedness predicate is applied only to regions $r_1, \ldots, r_n$ that are known to be pairwise disjoint, then it does not matter how many times this predicate occurs in the formula: satisfiability is still in PSpace.

The first main ingredient of our proofs is representation theorems allowing us to work with Aleksandrov topological spaces rather than arbitrary ones. Such spaces can be represented by Kripke frames with quasi-ordered accessibility relations. Topological connectedness in these frames corresponds to the graph-theoretic connectedness in the (non-directed) graphs induced by the accessibility relations. Based on this observation, one can prove the upper bounds in a more or less standard way using known techniques from modal and description logic. The lower bounds are much more involved and unexpected. They can be regarded as the main contribution of this paper.

2 Topological Logics

All our spatial logics are interpreted over *topological spaces*. Given such a space T and a set $X \subseteq T$, we denote by X° the *interior* of X in T and by X^- its *closure*. As usual in spatial KR&R, by a *region* of T we understand any *regular*

closed subset of T, i.e., any $X \subseteq T$ with $X = X^{\circ -}$. Denote by $\boldsymbol{RC}(T)$ the set of all regular closed subsets of T. It is known that $\boldsymbol{RC}(T)$ is a Boolean algebra with top and bottom elements given by T and $\emptyset$, Boolean operations $\cdot, -$ given by $X \cdot Y = (X \cap Y)^{\circ -}$ and $-X = (\overline{X})^{\circ -}$, and Boolean order $\leq$ by the relation $\subseteq$. Let $\mathcal{R} = \{r_i \mid i < \omega\}$ be a set of *region variables*. A *regular topological model over* T is a pair $\mathfrak{M} = (T, \cdot^{\mathfrak{M}})$, where $\cdot^{\mathfrak{M}}$ is a map from $\mathcal{R}$ to $\boldsymbol{RC}(T)$. Our minimal spatial logic, called $\mathcal{B}$, is defined as follows. The set of $\mathcal{B}$*-terms* is given by:

$$\tau \quad ::= \quad r_i \quad | \quad -\tau \quad | \quad \tau_1 \cdot \tau_2.$$

We abbreviate $-((-\tau_1) \cdot (-\tau_2))$ by $\tau_1 + \tau_2$, $r_0 \cdot (-r_0)$ by $\mathbf{0}$, and $-\mathbf{0}$ by $\mathbf{1}$. The set of $\mathcal{B}$*-formulas* is defined by:

$$\varphi \quad ::= \quad \tau_1 = \tau_2 \quad | \quad \neg\varphi \quad | \quad \varphi_1 \wedge \varphi_2.$$

Given a model $\mathfrak{M}$, the *extension* $\tau^{\mathfrak{M}}$ of a $\mathcal{B}$-term τ in $\mathfrak{M}$ is defined inductively by the equations $(-\tau)^{\mathfrak{M}} = (\overline{\tau^{\mathfrak{M}}})^{\circ -}$ and $(\tau_1 \cdot \tau_2)^{\mathfrak{M}} = (\tau_1^{\mathfrak{M}} \cap \tau_2^{\mathfrak{M}})^{\circ -}$, where $\overline{X} = T \setminus X$. The *truth-relation* for $\mathcal{B}$-formulas is defined by setting $\mathfrak{M} \models \tau_1 = \tau_2$ iff $\tau_1^{\mathfrak{M}} = \tau_2^{\mathfrak{M}}$, and interpreting the Boolean connectives $\neg$ and $\wedge$ in the standard way. We say that a formula φ is *satisfiable* (over a topological space T) if $\mathfrak{M} \models \varphi$, for some model $\mathfrak{M} = (T, \cdot^{\mathfrak{M}})$. Topologically, the logic $\mathcal{B}$ is quite poor: every satisfiable $\mathcal{B}$-formula φ is satisfied in a discrete topological space. In fact, it as expressive as the modal logic $\mathcal{S}5$, with $\tau = \mathbf{1}$ playing the role of the $\mathcal{S}5$-box.

The logic $\mathcal{C}$ extends $\mathcal{B}$ with the binary *contact* relation C due to Whitehead [22]. Specifically, $\mathcal{C}$*-formulas* are defined in the same way as the $\mathcal{B}$-formulas, except that we have the additional clause

$$\varphi \quad ::= \quad \ldots \quad | \quad C(\tau_1, \tau_2) \quad | \quad \ldots,$$

where τ_1 and τ_2 are $\mathcal{B}$-terms. The intended meaning of $C(\tau_1, \tau_2)$ is as expected: $\mathfrak{M} \models C(\tau_1, \tau_2)$ iff $\tau_2^{\mathfrak{M}} \cap \tau_2^{\mathfrak{M}} \neq \emptyset$, that is τ_1 is in contact with τ_2 in $\mathfrak{M}$. (It is to be noted that we may have $\mathfrak{M} \models \big(\tau_1 \cdot \tau_2 = \mathbf{0}\big) \wedge C(\tau_1, \tau_2)$.) Unlike $\mathcal{B}$, the logic $\mathcal{C}$ can express a number of important topological relationships between regions, e.g., all the $\mathcal{RCC}$-8 relations.

Finally, we define the well-known modal logic $\mathcal{S}4_u$ which can be regarded as a spatial logic in view of the topological interpretation of $\mathcal{S}4$ due to McKinsey and Tarski [13]. As $\mathcal{S}4_u$ is expressive enough to define the property of being regular closed, we take a new set $\mathcal{V} = \{v_i \mid i < \omega\}$ of *set variables* and interpret them by *arbitrary* sets of topological spaces. The $\mathcal{S}4_u$*-terms* are given by

$$\tau \quad ::= \quad v_i \quad | \quad \overline{\tau} \quad | \quad \tau_1 \cap \tau_2 \quad | \quad \tau^{\circ}.$$

We abbreviate $\overline{(\overline{\tau}^{\circ})}$ by τ^{-}, $\overline{(\overline{\tau_1} \cap \overline{\tau_2})}$ by $\tau_1 \cup \tau_2$, $v_0 \cap \overline{v_0}$ by $\mathbf{0}$, and $\overline{\mathbf{0}}$ by $\mathbf{1}$. The $\mathcal{S}4_u$*-formulas* are defined in the same way as $\mathcal{B}$-formulas.

In a *topological model* $\mathfrak{M} = (T, \cdot^{\mathfrak{M}})$ for $\mathcal{S}4_u$, $\cdot^{\mathfrak{M}}$ is a map from $\mathcal{V}$ to 2^T. The *extension* $\tau^{\mathfrak{M}}$ of a term τ in $\mathfrak{M}$ is defined inductively by the equations:

$$(\overline{\tau})^{\mathfrak{M}} = \overline{(\tau^{\mathfrak{M}})}, \qquad (\tau_1 \cap \tau_2)^{\mathfrak{M}} = \tau_1^{\mathfrak{M}} \cap \tau_2^{\mathfrak{M}}, \qquad (\tau^{\circ})^{\mathfrak{M}} = (\tau^{\mathfrak{M}})^{\circ}.$$

And the truth-relation for $\mathcal{S}4_u$-formulas is defined in the same way as for $\mathcal{B}$-formulas. Note that both $\mathcal{B}$ and $\mathcal{C}$ can be regarded as proper fragments of $\mathcal{S}4_u$.

3 Topological Logics with Connectedness

Recall that a topological space T is *connected* just in case it is not the union of two non-empty, disjoint, open sets; a subset $X \subseteq T$ is *connected in* T just in case either it is empty, or the topological space X (with the subspace topology) is connected. If $X \subseteq T$, a maximal connected subset of X is called a (*connected*) *component* of X. Every set X has at least one component, and a set is connected just in case it has at most one component. The $\mathcal{S}4_u$-formula

$$(v_1 \neq \mathbf{0}) \ \wedge \ (v_2 \neq \mathbf{0}) \ \wedge \ (v_1 \cup v_2 = \mathbf{1}) \ \wedge \ (v_1^- \cap v_2 = \mathbf{0}) \ \wedge \ (v_1 \cap v_2^- = \mathbf{0})$$

is satisfiable in a topological space T iff T is not connected; it was used in [21] to axiomatize the logic (in the standard language of $\mathcal{S}4_u$) of connected spaces.

We now extend the logics $\mathcal{B}$, $\mathcal{C}$ and $\mathcal{S}4_u$ with the connectedness predicate $c(\cdot)$ and denote the resulting languages by $\mathcal{B}c$, $\mathcal{C}c$ and $\mathcal{S}4_uc$, respectively. Their formulas are defined as before, except that we now have the additional clause:

$$\varphi \quad ::= \quad \ldots \quad | \quad c(\tau) \quad | \quad \ldots.$$

The meaning of $c(\tau)$ in a model $\mathfrak{M} = (T, \cdot^{\mathfrak{M}})$ is as follows: $\mathfrak{M} \models c(\tau)$ iff $\tau^{\mathfrak{M}}$ is connected in T. For example, most textbooks on general topology prove the following facts: (i) the union of two intersecting, connected sets is connected; (ii) any set sandwiched between a connected set and its closure is itself connected. These facts are expressible as the following $\mathcal{S}4_uc$-validities:

$$c(v_1) \ \wedge \ c(v_2) \ \wedge \ (v_1 \cap v_2 \neq \mathbf{0}) \ \rightarrow \ c(v_1 \cup v_2),$$
$$c(v_1) \ \wedge \ (v_1 \subseteq v_2) \ \wedge \ (v_2 \subseteq v_1^-) \ \rightarrow \ c(v_2).$$

One can increase the expressive power of the connectedness predicate $c(\tau)$ by generalizing it to the 'counting' predicates $c^{\leq k}(\tau)$, $1 \leq k < \omega$, which state that τ has at most k connected components. We denote the languages with such predicates by $\mathcal{B}cc$, $\mathcal{C}cc$ and $\mathcal{S}4_ucc$. Their formulas are defined in the same way as before, except that we have the additional clause, where $1 \leq k < \omega$:

$$\varphi \quad ::= \quad \ldots \quad | \quad c^{\leq k}(\tau) \quad | \quad \ldots.$$

The meaning of $c^{\leq k}(\tau)$ is as follows: $\mathfrak{M} \models c^{\leq k}(\tau)$ iff $\tau^{\mathfrak{M}}$ has at most k components in T. We write $\neg c^{\leq k}(\tau)$ as $c^{\geq k+1}(\tau)$ and abbreviate $c^{\leq 1}(\tau)$ by $c(\tau)$. Thus, we may regard $\mathcal{S}4_uc$ as a sub-language of $\mathcal{S}4_ucc$. The numerical superscripts k in $c^{\leq k}$ are assumed to be coded in *binary*.

Note that for each $\mathcal{S}4_ucc$-formula φ one can construct an equi-satisfiable $\mathcal{S}4_uc$-formula φ' using the observation that $c^{\leq k}(\tau)$ can be replaced by (1) if it occurs positively in φ and by (2) if the occurrence is negative, where

$$\Big(\tau = \bigcup_{1 \leq i \leq k} v_i\Big) \ \wedge \ \bigwedge_{1 \leq i \leq k} c(v_i), \tag{1}$$

$$\Big(\tau = \bigcup_{1 \leq i \leq k+1} v_i\Big) \ \wedge \ \bigwedge_{1 \leq i \leq k+1} (v_i \neq \mathbf{0}) \ \wedge \ \bigwedge_{1 \leq i < j \leq k+1} (\tau \cap v_i^- \cap v_j^- = \mathbf{0}) \tag{2}$$

with fresh $v_1, \ldots, v_k$. Note, however, that these $\mathcal{S}4_uc$-formulas are exponentially larger than the literals they replace.

4 Computational Complexity

There are two known complexity results for the spatial logics with connectedness constraints introduced above. According to [15], satisfiability of $\mathcal{S}4_u cc$-formulas is NExpTime-complete, which gives the NExpTime upper bound for all of these logics. On the other hand, it follows from [23] that $\mathcal{C}c$ is PSpace-hard (more precisely, satisfiability of $\mathcal{C}$-formulas in *connected* spaces is PSpace-complete).

We begin by showing that, as far as satisfiability is concerned, we can restrict attention to topological spaces of a special kind. Recall that a topological space is called an *Aleksandrov space* if arbitrary (not only finite) intersections of open sets are open. Aleksandrov spaces can be characterized in terms of *Kripke frames* $\mathfrak{F} = (W, R)$, where $W \neq \emptyset$ and R is a transitive and reflexive relation (i.e., a *quasi-order*) on W. Every such $\mathfrak{F}$ induces the interior operator $\cdot^\circ_{\mathfrak{F}}$ on W:

$$X^\circ_{\mathfrak{F}} \;=\; \{x \in X \mid \forall y \in W\ (xRy \to y \in X)\}, \quad \text{for every } X \subseteq W.$$

It is well-known [2] that the resulting topological space is Aleksandrov and, conversely, every Aleksandrov space is induced by a quasi-order. Topological models over Aleksandrov spaces will be called *Aleksandrov models*. Note that the Aleksandrov space induced by $\mathfrak{F} = (W, R)$ is connected iff $\mathfrak{F}$ is *connected* as a non-directed graph, that is, between any two points $x, y \in W$ there is a path along the relation $R \cup R^{-1}$, where R^{-1} is the inverse of R. This observation is used implicitly throughout this paper. It is shown in [15] that $\mathcal{S}4_u cc$ is complete w.r.t. finite Aleksandrov models; this is a consequence of the following lemma.

Lemma 1 ([13,15]). (i) *For every $\mathcal{S}4_u cc$-formula φ and every $\mathfrak{M} = (T, \cdot^{\mathfrak{M}})$ there exist an Aleksandrov model $\mathfrak{A} = (T_A, \cdot^{\mathfrak{A}})$ with $|T_A| \leq 2^{|\varphi|}$ and a continuous function $f\colon T \to T_A$ such that, for every sub-term τ of φ, $\tau^{\mathfrak{A}} = f(\tau^{\mathfrak{M}})$.*

(ii) *Every $\mathcal{S}4_u cc$-formula φ can be transformed (in* LogSpace*) into an $\mathcal{S}4_u cc$-formula φ' such that it has no negative occurrences of $c^{\leq k}(\tau)$, $|\varphi'|$ is polynomial in $|\varphi|$, and both φ and φ' are satisfiable over the same topological spaces.*

According to the next lemma, satisfiable $\mathcal{C}cc$-formulas can be satisfied in Aleksandrov models based on partial orders (W, R) *of depth* 1, i.e., R is the reflexive closure of a subset of $W_1 \times W_0$, where W_i is the set of points *of depth* i; see Fig. 1. Such frames and models are called *quasi-saws* and *quasi-saw models*.

Lemma 2. *For every finite Aleksandrov model $\mathfrak{A} = (T_A, \cdot^{\mathfrak{A}})$, with T_A induced by (W, R_A), there is a quasi-saw model $\mathfrak{B} = (T_B, \cdot^{\mathfrak{B}})$ such that T_B is induced by (W, R_B) with $R_B \subseteq R_A$ and, for every $\mathcal{B}$-term τ,* (i) $\tau^{\mathfrak{B}} = \tau^{\mathfrak{A}}$, *and* (ii) τ *has the same number of components in $\mathfrak{A}$ and $\mathfrak{B}$.*

Proof. Let W_0 be the set of points from final clusters in (W, R_A), i.e., $W_0 = \{v \in W \mid vR_Au \text{ implies } uR_Av, \text{ for all } u \in W\}$. In every final cluster $C \subseteq W_0$ with $|C| \geq 2$ we select a point and denote by U the set of all selected points. Then we set $V_0 = W_0 \setminus U$ and $V_1 = W \setminus V_0$, and define R_B to be the reflexive closure of $R_A \cap (V_1 \times V_0)$. Clearly, (W, R_B) is a quasi-saw, with V_0 and V_1 being the sets of points of depth 0 and 1, respectively. For each variable r_i, let $r_i^{\mathfrak{B}} = r_i^{\mathfrak{A}}$. Claims (i) and (ii) are proved by induction on the construction of τ. ❑

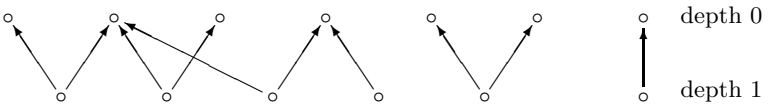


Fig. 1. Quasi-saw

4.1 Upper Complexity Bounds

We first prove the ExpTime-upper bound for $\mathcal{S}4_u c$ and a PSpace-upper bound for certain fragments of $\mathcal{S}4_u c$. To start with, we transform a given $\mathcal{S}4_u c$-formula φ into negation normal form (NNF$^+$) in the following way. First, we push negation $\neg$ inward to atoms $\tau_1 = \tau_2$ and $c(\tau)$, then use (2), for $k = 1$, to get rid of negative occurrences of $c(\tau)$, and finally replace each $c(\tau)$ with $(c(\tau) \wedge (\tau \neq \mathbf{0})) \vee (\tau = \mathbf{0})$, and each $(\tau_1 = \tau_2)$ with $(\tau_1 \cap \overline{\tau}_2 = \mathbf{0}) \wedge (\overline{\tau}_1 \cap \tau_2 = \mathbf{0})$.

Every $\mathcal{S}4_u c$-formula φ in NNF$^+$ is clearly equivalent to a disjunction $\bigvee \Psi_\varphi$, where each $\psi \in \Psi_\varphi$ is a conjunction of the form

$$\psi \quad = \quad \bigwedge_{i=1}^{l} (\rho_i = \mathbf{0}) \ \wedge \ \bigwedge_{i=1}^{m} (\tau_i \neq \mathbf{0}) \ \wedge \ \bigwedge_{i=1}^{k} \big(c(\sigma_i) \wedge (\sigma_i \neq \mathbf{0})\big) \tag{3}$$

such that each atom of φ occurs either positively or negatively in ψ. For any such conjunction, it is decidable in polynomial time (in $|\varphi|$) whether it is in Ψ_φ.

Theorem 1. *Satisfiability of $\mathcal{S}4_u c$-formulas is in* ExpTime.

Proof. The proof is by reduction to the satisfiability problem for propositional dynamic logic ($\mathcal{PDL}$) with converse and nominals, which is known to be ExpTime-complete [9, Section 7.3]. Let ψ be as in (3). Take two atomic programs α and β and, for each σ_i, a nominal ℓ_i. For a term τ, denote by $\tau^\dagger$ the $\mathcal{PDL}$-formula obtained by replacing in τ, recursively, each sub-term ϑ° with $[\alpha^*]\vartheta$. Thus α^* simulates the $\mathcal{S}4$-accessibility relation, and the universal box will be simulated by $[\gamma]$, where $\gamma = (\beta \cup \beta^- \cup \alpha \cup \alpha^-)^*$. Consider now the formula ψ'

$$\bigwedge_{i=1}^{l} [\gamma]\neg\rho_i^\dagger \ \wedge \ \bigwedge_{i=1}^{m} \langle\gamma\rangle\tau_i^\dagger \ \wedge \ \bigwedge_{i=1}^{k} \Big(\langle\gamma\rangle(\ell_i \wedge \sigma_i^\dagger) \ \wedge \ [\gamma](\sigma_i^\dagger \rightarrow \langle(\alpha \cup \alpha^-; \sigma_i^\dagger?)^*\rangle\ell_i)\Big).$$

It is not hard to see that ψ' is satisfiable iff ψ is satisfiable: the first conjunct of ψ' states that all ρ_i are empty, the second that all τ_i are non-empty, the third states that each σ_i holds at a point where ℓ_i holds and that from each σ_i-point there is a path (along $\alpha \cup \alpha^-$) to ℓ_i which lies entirely within σ_i. ❑

Denote by $\mathcal{S}4_u c^1$ the set of $\mathcal{S}4_u c$-formulas in NNF$^+$ with *at most one* occurrence of an atom of the form $c(\tau)$.

Theorem 2. *Satisfiability of $\mathcal{S}4_u c^1$-formulas is in* PSpace.

Proof. We sketch a nondeterministic PSPACE algorithm. Let φ be in NNF$^+$. Guess a ψ of the form (3) and check whether it is in Ψ_φ. Now check whether ψ is satisfiable: if ψ does not contain a conjunct of the form $c(\sigma) \wedge (\sigma \neq \mathbf{0})$, then a standard satisfiability checking algorithm for $\mathcal{S}4_u$ is applied. If it contains $c(\sigma) \wedge (\sigma \neq \mathbf{0})$, then the algorithm proceeds as follows. Let $\tau_0 = \bigcap_{i=1}^{l} \rho_i$. Set $\mathbf{B} = \{\overline{\tau_0}^\circ\} \cup \{\tau, \overline{\tau} \mid \tau \in \mathit{term}(\psi)\}$, where $\mathit{term}(\varphi)$ is the set of all sub-terms of ψ. A subset $\mathfrak{t}$ of $\mathbf{B}$ is called a *type for* ψ if $\overline{\tau_0}^\circ \in \mathfrak{t}$ and $\tau \in \mathfrak{t}$ iff $\overline{\tau} \notin \mathfrak{t}$, for all $\overline{\tau} \in \mathbf{B}$.

Now, guess a type $\mathfrak{t}_\sigma$ containing σ and start $m+1$ $\mathcal{S}4$-tableau procedures with inputs $\tau_1 \cap \overline{\tau_0}^\circ$, $\tau_2 \cap \overline{\tau_0}^\circ, \dots, \tau_m \cap \overline{\tau_0}^\circ$, and $\bigcap \mathfrak{t}_\sigma \cap \overline{\tau_0}^\circ$ in the usual way expanding branch-by-branch, recovering the space once branches are checked. We may as well assume that the nodes of these tableaux are types. Suppose $\mathfrak{t}$ is a type occurring in one of them. If $\sigma \in \mathfrak{t}$, it suffices to check that $\mathfrak{t}$ can be connected by a path of $\leq 2^{|\psi|}$ points in σ to $\mathfrak{t}_\sigma$. To complete the proof we present a subroutine which, given types $\mathfrak{t}_0, \mathfrak{t}_1 \ni \sigma$ and $d \geq 0$, checks, in PSPACE, whether $\mathfrak{t}_0$ and $\mathfrak{t}_1$ can be connected by a path of $\leq 2^d$ points in σ to $\mathfrak{t}_\sigma$.

Subroutine: If $d = 0$, we check that $\mathfrak{t}_0$ and $\mathfrak{t}_1$ can be made accessible one direction or the other. If $d > 0$, we guess a type $\mathfrak{t}$ with $\sigma \in \mathfrak{t}$ that represents the half-way point between $\mathfrak{t}_0$ and $\mathfrak{t}_1$. First we check that $\mathfrak{t}$ is an allowable type by constructing an $\mathcal{S}4$-tableau with root $\mathfrak{t}$. The tableau can be discarded after completion: although it may contain types $\mathfrak{t}'$ with $\sigma \in \mathfrak{t}'$, these type can never threaten the connectedness of σ, since they are all accessible from the root $\mathfrak{t}$ of the tableau (the $\mathcal{S}4$ accessibility relation is transitive!), and so are connected to both $\mathfrak{t}_0$ and $\mathfrak{t}_1$ anyway. Then the subroutine calls itself recursively with parameters $(\mathfrak{t}_0, \mathfrak{t}, d-1)$ and $(\mathfrak{t}, \mathfrak{t}_1, d-1)$. Completing this recursive procedure requires at most d items to be placed on the stack. ❑

Observe that the argument above shows that satisfiability of formulas φ in NNF$^+$ with conjuncts $\bigwedge_{i=1}^{k} c(\tau_i)$ such that $(\tau_i^- \cap \tau_j^- = \mathbf{0})$, $i \neq j$, are conjuncts of φ, is decidable in PSPACE as well.

4.2 Lower Complexity Bounds

We first prove the matching lower bound for $\mathcal{C}c$. Observe that when constructing a model for an $\mathcal{S}4_u c^1$-formula with one positive occurrence of $c(\tau)$, we can check 'connectivity' of two τ-points by an (exponentially long) path using a PSPACE-algorithm because it is not necessary to keep in memory all the points on the path. However, if two statements $c(\tau_1)$ and $c(\tau_2)$ have to be satisfied, then, while connecting two τ_1-points using a path, one has to check whether the τ_2-points on that path can be connected by a path, which, in turn, can contain another τ_1-point, and so on. The crucial idea in the proof below is simulating infinite binary (*non-transitive*) trees using quasi-saws. Roughly, the construction is as follows. We start by representing the root v_0 of the tree as a point also denoted by v_0 (see Fig. 2), which is forced to be connected to an auxiliary point z by means of some $c(\tau_0)$. On the connecting path from v_0 to z we represent the two successors v_1 and v_2 of the root, which are forced to be connected in their turn to z by some other $c(\tau_1)$. On each of the two connecting paths, we again

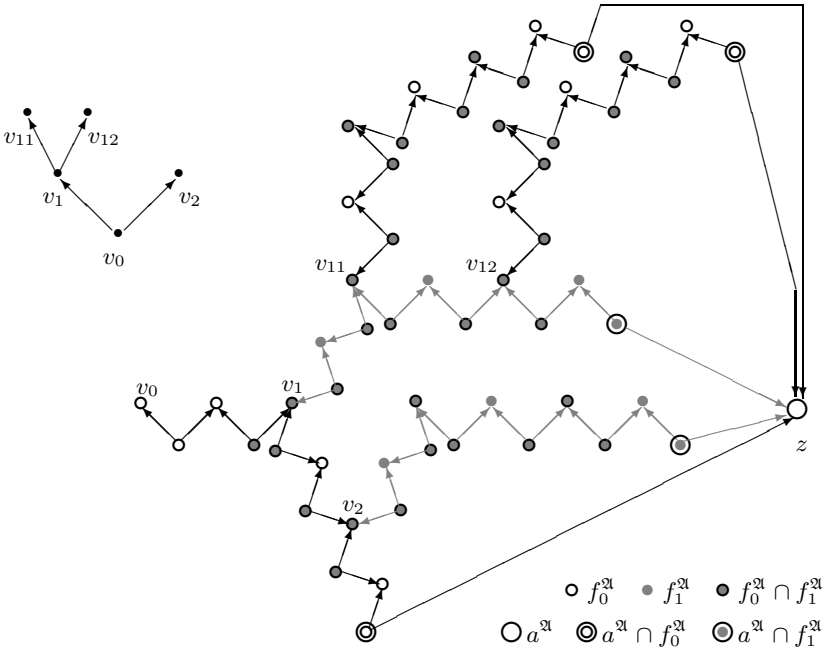


Fig. 2. First 4 steps of encoding the full binary tree using 7-saws

take two points representing the successors of v_1 and v_2, respectively. We treat these four points in the same way as v_0, reusing $c(\tau_0)$, and proceed *ad infinitum* alternating between τ_0 and τ_1 when forcing the paths which generate the required successors. Of course, we also have to pass certain information from a node to its two successors (say, if $\Diamond\psi$ holds in the node, then ψ holds in one of its successors). Such information can be propagated along connected regions. Note now that all points are connected to z. To distinguish between the information we have to pass from distinct nodes of even (respectively, odd) level to their successors, we have to use *two* connectedness formulas of the form $c(f_i + a)$, $i = 0, 1$, in such a way that the f_i points form initial segments of the paths to z and a contains z. The f_i-segments are then used locally to pass information from a node to its successors without conflict. We now present the reduction in more detail.

Theorem 3. *Satisfiability of $\mathcal{C}c$-formulas is* ExpTime*-hard.*

Proof. The proof is by reduction of the following problem. Denote by $\mathcal{D}_2^f$ the bimodal logic (with $\Box_1$ and $\Box_2$) determined by Kripke models based on the full infinite binary tree $\mathfrak{G} = (V, R_1, R_2)$ with *functional* accessibility relations R_1 and R_2. Consider the *global consequence relation* $\models_2^f$ defined as follows: $\chi \models_2^f \psi$ iff $\mathfrak{K} \models \chi$ implies $\mathfrak{K} \models \psi$, for every Kripke model $\mathfrak{K}$ based on $\mathfrak{G}$. Using standard modal logic technique one can show ExpTime-hardness of this global

consequence relation. We construct a $\mathcal{C}c$-formula $\Phi(\chi,\psi)$, for any $\mathcal{D}_2^f$-formulas χ, ψ, such that (i) $|\Phi(\chi,\psi)|$ is polynomial in $|\chi|+|\psi|$ and (ii) $\Phi(\chi,\psi)$ is satisfiable iff $\chi \not\models_2^f \psi$. While constructing $\Phi(\chi,\psi)$, we will assume that $\mathfrak{A}$ is a quasi-saw model induced by (W,R) and W_0 is the set of points of depth 0 in (W,R).

Let $sub(\chi,\psi)$ be the closure under single negation of the set of subformulas of χ,ψ. For each $\varphi \in sub(\chi,\psi)$ we take a fresh variable q_φ, and for $\Box_i\varphi \in sub(\chi,\psi)$ and $j=0,1$, we fix fresh variables $m_\varphi^{i,j}$ and $m_{\neg\varphi}^{i,j}$. We also need fresh variables s_j^i, for $j=0,1$ and $0\le i\le 6$. Let $d = s_0^0 + s_1^0$. Intuitively, d simulates the domain of the binary tree, where s_0^0 and s_1^0 stand for nodes with even and, respectively, odd distance from the root. Suppose that the following $\mathcal{C}c$-formulas hold in $\mathfrak{A}$

$$\big(s_0^6 = s_1^6\big) \quad\wedge\quad \big(s_0^6 \neq \mathbf{0}\big) \quad\wedge\quad c(f_0 + s_0^6) \quad\wedge\quad c(f_1 + s_1^6), \tag{4}$$

$$\bigwedge_{0\le k<k'\le 6} (s_j^k \cdot s_j^{k'} = \mathbf{0}) \quad\wedge\quad \bigwedge_{\substack{0\le k<k'\le 6\\ |k-k'|>1}} \neg C(s_j^k, s_j^{k'}), \tag{5}$$

where $f_j = s_j^0 + s_j^1 + s_j^2 + s_j^3 + s_j^4 + s_j^5$, for $j=0,1$. (Note that s_0^6 and s_1^6 play the role of a in the explanation above; see Fig. 2.) It follows that, for $j=0,1$, if there is a point $x_0 \in (s_j^0)^{\mathfrak{A}} \cap W_0$ then there is a (not necessarily unique) sequence of points x_1,x_2,x_3,x_4,x_5 from the same connected component of $f_j^{\mathfrak{A}}$ such that $x_i \in (s_j^i)^{\mathfrak{A}} \cap W_0$, $1\le i\le 5$. Points x_2 and x_4 will be used to construct similar sequences for the two successors of the node represented by x_0: if (4)–(5) and

$$s_0^{2i} \;\le\; s_1^0 \qquad \text{and} \qquad s_1^{2i} \;\le\; s_0^0, \qquad \text{for } i=1,2, \tag{6}$$

hold in $\mathfrak{A}$ and $x_0 \in (s_j^0)^{\mathfrak{A}} \cap W_0$, then one can recover from $\mathfrak{A}$ the infinite binary tree with the root at x_0. The formula

$$(q_{\neg\psi} \cdot s_0^0 \neq \mathbf{0}) \quad\wedge\quad (d \;\le\; q_\chi) \tag{7}$$

ensures then that there is $x_0 \in (s_j^0)^{\mathfrak{A}} \cap W_0$, the root of the tree, in which ψ holds, and χ holds everywhere in the tree, while the formulas

$$d \cdot q_{\neg\varphi} = d \cdot (-q_\varphi), \qquad d \cdot q_{\varphi_1\wedge\varphi_2} = d \cdot (q_{\varphi_1} \cdot q_{\varphi_2}), \tag{8}$$

for all $\neg\varphi, \varphi_1\wedge\varphi_2 \in sub(\chi,\psi)$, capture the meaning of the Boolean connectives from $sub(\chi,\psi)$ relativized to d. The formulas

$$\neg C(f_j \cdot m_\varphi^{i,j}, f_j \cdot m_{\neg\varphi}^{i,j}), \tag{9}$$

$$(s_j^0 \cdot q_{\Box_i\varphi} \le m_\varphi^{i,j}) \quad\wedge\quad (m_\varphi^{i,j} \cdot s_j^{2i} \;\le\; q_\varphi), \tag{10}$$

$$(s_j^0 \cdot q_{\neg\Box_i\varphi} \le m_{\neg\varphi}^{i,j}) \quad\wedge\quad (m_{\neg\varphi}^{i,j} \cdot s_j^{2i} \;\le\; q_{\neg\varphi},), \tag{11}$$

for all $\Box_i\varphi \in sub(\chi,\psi)$ and $j=0,1$, are used to propagate information regarding $\Box_i\varphi$ along the connected components of f_j using the markers $m_\varphi^{i,j}$ and $m_{\neg\varphi}^{i,j}$.

We define $\Phi(\chi,\psi)$ to be the conjunction of all the above formulas. Clearly, $|\Phi(\chi,\psi)|$ is polynomial in $|\chi|+|\psi|$ and contains only two occurrences of the connectedness predicate in (4).

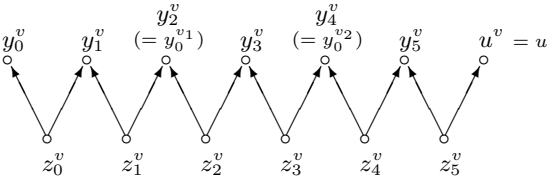


Fig. 3. A 7-saw for v

Conversely, suppose that $\mathfrak{K}$ is a model for $\mathcal{D}_2^f$ based on the full infinite binary tree $\mathfrak{G} = (V, R_1, R_2)$ with root v_0 and such that $\mathfrak{K} \models \chi$ and $\mathfrak{K}, v_0 \not\models \psi$. We construct a quasi-saw model $\mathfrak{A}$ satisfying $\Phi(\chi, \psi)$ by induction (as in Fig. 2) using infinitely many copies of the 7-saw shown in Fig. 3. For each node v of $\mathfrak{G}$, we take a fresh 7-saw $\mathfrak{S}^v = (S^v, R^v)$, where $S^v = \{y_i^v, z_i^v, u^v \mid 0 \leq i \leq 5\}$, $z_i^v R^v y_i^v$, $z_i^v R^v y_{i+1}^v$, for $0 \leq i \leq 5$, and $z_5^v R^v u^v$, and identify the following points: $y_2^v = y_0^{v_1}$, $y_4^v = y_0^{v_2}$, $u^{v_1} = u^{v_2} = u^v$, if v_1 and v_2 are the R_1- and R_2-successors of v. The assignment is left to the reader.

Note that y_1^v and y_3^v are required to make the set of points in $f_j^{\mathfrak{A}}$ representing a node v of $\mathfrak{G}$ disconnected from the subset of $f_j^{\mathfrak{A}}$ representing another node v' of $\mathfrak{G}$ and thus satisfy (9); y_5^v are required to satisfy the last conjunct of (5). ❑

We now consider the lower complexity bound for $\mathcal{C}$ with constraints on the number of connected components.

Theorem 4. *Satisfiability of $\mathcal{C}cc$-formulas is* NExpTime*-hard.*

Proof. The proof is by reduction of the NExpTime-complete $2^d \times 2^d$ *tiling problem*: Given $d < \omega$, a finite set $\mathcal{T}$ of *tile types*—i.e., 4-tuples of colours $T = (\mathit{left}(T), \mathit{right}(T), \mathit{up}(T), \mathit{down}(T))$—and a $T_0 \in \mathcal{T}$, decide whether $\mathcal{T}$ can tile the $2^d \times 2^d$ grid in such a way that T_0 is placed onto $(0, 0)$. In other words, the problem is to decide whether there is a function f from $\{(i, j) \mid i, j < 2^d\}$ to $\mathcal{T}$ such that $\mathit{up}(f(i, j)) = \mathit{down}(f(i, j + 1))$, for all $i < 2^d$, $j < 2^d - 1$, $\mathit{right}(f(i, j)) = \mathit{left}(f(i + 1, j))$, for all $i < 2^d - 1$, $j < 2^d$, $f(0, 0) = T_0$. We construct a $\mathcal{C}cc$-formula $\varphi_{\mathcal{T},d}$ such that (i) $|\varphi_{\mathcal{T},d}|$ is polynomial in $|\mathcal{T}|$ and d and (ii) $\varphi_{\mathcal{T},d}$ is satisfiable iff $\mathcal{T}$ tiles the $2^d \times 2^d$ grid, with T_0 being placed onto $(0, 0)$. While constructing $\varphi_{\mathcal{T},d}$, we will assume that $\mathfrak{A}$ is a quasi-saw model induced by (W, R) and W_0 is the set of points of depth 0 in (W, R).

We partition all points of W_0 with the help of a pair of variable triples B_X^0, B_X^1, B_X^2 and B_Y^0, B_Y^1, B_Y^2. Suppose that the formulas, for $0 \leq \ell < 3$,

$$B_X^0 + B_X^1 + B_X^2 = \mathbf{1}, \qquad B_X^\ell \cdot B_X^{\ell \oplus_3 1} = \mathbf{0} \tag{12}$$

and their Y-counterparts hold in $\mathfrak{A}$, where $\oplus_3$ denotes addition modulo 3. Then every point in W_0 is in exactly one of the $(B_X^\ell)^{\mathfrak{A}}$ and exactly one of the $(B_Y^\ell)^{\mathfrak{A}}$.

To encode coordinates of the tiles in binary, we take a pair of variables X_j and Y_j, for each $d \geq j \geq 1$. For $0 \leq n < 2^d$, let n_X be the $\mathcal{B}$-term $X'_d \cdot X'_{d-1} \cdot \dots \cdot X'_1$, where $X'_j = X_j$ if the jth bit in the binary representation of n is 1, and $X'_j = -X_j$ otherwise. For a point $u \in W_0$, we denote by $X(u)$ the binary d-bit number n,

called the X-*value* of u, such that $u \in n_X^{\mathfrak{A}}$; the jth bit of $X(u)$ is denoted by $X_j(u)$. The term n_Y, the Y-*value* $Y(u)$ of u and its jth bit $Y_j(u)$ are defined analogously. For a point u of depth 0 we write $\mathsf{coor}(u)$ for $(X(u), Y(u))$. We will use the variables X_i and Y_j to generate the $2^d \times 2^d$ grid, which consists of pairs (i_X, j_Y), for $0 \le i, j < 2^d$. Consider the following formulas, for $0 \le \ell < 3$,

$$\neg C(X_k \cdot B_X^\ell,\ (-X_k) \cdot B_X^\ell), \qquad d \ge k \ge 1, \tag{13}$$

$$\neg C(X_j \cdot (-X_k) \cdot B_X^\ell,\ (-X_j) \cdot B_X^{\ell \oplus_3 1}), \qquad d \ge j > k \ge 1, \tag{14}$$

$$\neg C((-X_j) \cdot (-X_k) \cdot B_X^\ell,\ X_j \cdot B_X^{\ell \oplus_3 1}), \qquad d \ge j > k \ge 1, \tag{15}$$

$$\neg C((-X_k) \cdot X_{k-1} \cdot \ \cdots \ \cdot X_1 \cdot B_X^\ell,\ \ (-X_k) \cdot B_X^{\ell \oplus_3 1}), \qquad d \ge k > 1, \tag{16}$$

$$\neg C((-X_k) \cdot X_{k-1} \cdot \ \cdots \ \cdot X_1 \cdot B_X^\ell,\ \ X_i \cdot B_X^{\ell \oplus_3 1}), \qquad d \ge k > i \ge 1, \tag{17}$$

$$\neg C(X_d \cdot \ \cdots \ \cdot X_1,\ (-X_d) \cdot \ \cdots \ \cdot (-X_1)), \tag{18}$$

the Y-counterparts of (13)–(18), and the following, for $d \ge j, k \ge 1$,

$$\neg C(X_j \cdot Y_k,\ (-X_j) \cdot (-Y_k)), \qquad \neg C((-X_j) \cdot Y_k,\ X_j \cdot (-Y_k)). \tag{19}$$

Given a point $v \in W_0$, denote by $\mathsf{4\text{-}nb}(v)$ the set which consists of $\mathsf{coor}(v)$ and its (at most four) neighbours in the $2^d \times 2^d$ grid. Suppose that $\mathfrak{A}$ satisfies all the formulas above. If $u, v \in W_0$ and zRu and zRv, for some $z \in W$, then $\mathsf{coor}(u) \in \mathsf{4\text{-}nb}(v)$. Moreover, (i) $X(v) = X(u) = n$ iff u and v are in the same component of $n_X^{\mathfrak{A}}$, and (ii) for each $m = -1, 0, 1$, $X(v) = X(u) + m$ iff $u \in (B_X^\ell)^{\mathfrak{A}}$ and $v \in (B_X^{\ell \oplus_3 m})^{\mathfrak{A}}$, for $\ell = 0, 1, 2$ (in particular, $X(u) = X(v)$ iff u and v are in the same connected component of $(B_X^\ell)^{\mathfrak{A}}$). Likewise for Y in place of X.

Suppose now that the following formulas are true in $\mathfrak{A}$ as well:

$$0_X \cdot 0_Y \neq \mathbf{0},\ (2^d - 1)_X \cdot (2^d - 1)_Y \neq \mathbf{0},\ c(0_X + (2^d - 1)_Y),\ c((2^d - 1)_X + 0_Y). \tag{20}$$

These constraints guarantee that in the connected set $(0_X + (2^d - 1)_Y)^{\mathfrak{A}}$ there are points $u_{(0,i)}$ and $u_{(i,2^d-1)}$, $0 \le i < 2^d$, such that $\mathsf{coor}(u_{(0,i)}) = (0, i)$ and $\mathsf{coor}(u_{(i,2^d-1)}) = (i, 2^d - 1)$. Similarly for the connected set $((2^d - 1)_X + 0_Y)^{\mathfrak{A}}$. This gives us the border of the $2^d \times 2^d$ grid we are after. And the constraints

$$c((-X_1) + 0_Y), \quad c(X_1 + 0_Y), \quad c(0_X + (-Y_1)), \quad c(0_X + Y_1) \tag{21}$$

ensure that we can find inner points of the grid. It is to be noted, however, that in general $u \neq v$ even if $\mathsf{coor}(u) = \mathsf{coor}(v)$. In other words, the constructed points do not necessarily form a proper $2^d \times 2^d$ grid. Let $\mathbf{b} = \big(X_1 \cdot (-Y_1)\big) + \big((-X_1) \cdot Y_1\big)$ and $\mathbf{w} = \big((-X_1) \cdot (-Y_1)\big) + \big(X_1 \cdot Y_1\big)$. Points in $\mathbf{b}^{\mathfrak{A}}$ and $\mathbf{w}^{\mathfrak{A}}$ can be thought of as *black* and *white* squares of a chessboard. Observe that if $u, v \in \mathbf{b}^{\mathfrak{A}} \cap W_0$ and $\mathsf{coor}(u) \neq \mathsf{coor}(v)$ then u and v cannot belong to the same component of $\mathbf{b}^{\mathfrak{A}}$. Thus, there are at least 2^{d-1} components in both $\mathbf{b}^{\mathfrak{A}}$ and $\mathbf{w}^{\mathfrak{A}}$. Our next constraints

$$c^{\le 2^{d-1}}(\mathbf{b}), \qquad c^{\le 2^{d-1}}(\mathbf{w}) \tag{22}$$

say that $\mathbf{b}^{\mathfrak{A}}$ and $\mathbf{w}^{\mathfrak{A}}$ have precisely 2^{d-1} components. In particular, if $u, v \in W_0$ belong to the same component of $\mathbf{b}^{\mathfrak{A}}$ then $\mathsf{coor}(u) = \mathsf{coor}(v)$. This gives a proper $2^d \times 2^d$ grid on which we encode the tiling conditions. The formulas

$$\sum_{T \in \mathcal{T}} T = \mathbf{1} \qquad \text{and} \qquad T \cdot T' = \mathbf{0}, \quad \text{for } T \neq T', \tag{23}$$

$$\neg C(B_X^{\ell} \cdot B_Y^{\ell'} \cdot T,\ B_X^{\ell} \cdot B_Y^{\ell'} \cdot T'), \quad \text{for } \ell, \ell' = 0, 1, 2 \quad \text{and} \quad T \neq T', \tag{24}$$

say that every point in W_0 is covered by precisely one tile and that all points in the same component of $(B_X^{\ell} \cdot B_Y^{\ell'})^{\mathfrak{A}}$ are covered by the same tile. That the colours of adjacent tiles match is ensured by

$$\neg C(B_X^{\ell} \cdot T,\ B_X^{\ell \oplus_3 1} \cdot T'), \qquad \text{for } T, T' \in \mathcal{T} \text{ with } \mathit{right}(T) \neq \mathit{left}(T'), \tag{25}$$

$$\neg C(B_Y^{\ell} \cdot T,\ B_Y^{\ell \oplus_3 1} \cdot T'), \qquad \text{for } T, T' \in \mathcal{T} \text{ with } \mathit{top}(T) \neq \mathit{bot}(T'). \tag{26}$$

Finally, we have to say that $(0, 0)$ is covered with T_0:

$$0_X \cdot 0_Y \leq T_0. \tag{27}$$

One can check that the conjunction $\varphi_{\mathcal{T},d}$ of these $\mathcal{C}cc$-formulas is as required. ❑

The ExpTime and NExpTime lower bounds for $\mathcal{B}c$ and $\mathcal{B}cc$ will be proved by reduction of satisfiability for $\mathcal{C}c$ and $\mathcal{C}cc$, respectively; that is, by eliminating occurrences of the predicate C in $\mathcal{C}c$- and $\mathcal{C}cc$-formulas. Clearly, two *connected* closed sets are in contact iff their union is connected; in other words, the formula $c(\tau_1) \wedge c(\tau_2) \to \big(C(\tau_1, \tau_2) \ \leftrightarrow \ c(\tau_1 + \tau_2)\big)$ is a $\mathcal{C}cc$-validity. However, this 'reduction' of C to c cannot be directly applied to our formulas since the arguments of the contact predicates in them are not necessarily connected. The next three lemmas show how to overcome this problem.

We write $\varphi[\psi]^+$ (or $\varphi[\psi]^-$) to indicate that φ contains a positive (respectively, negative) occurrence of ψ; then $\varphi[\chi]^+$ (or $\varphi[\chi]^-$) denotes the result of replacing this occurrence of ψ in φ by χ.

Lemma 3. *Let $\varphi[C(\tau_1, \tau_2)]^+$ be a $\mathcal{C}cc$-formula, and t, t_1, t_2 fresh variables. Then φ is equisatisfiable with the formula*

$$\varphi^* \quad = \quad \varphi[t = \mathbf{0}]^+ \quad \wedge \quad \big((t = \mathbf{0}) \quad \to \quad c(t_1 + t_2) \ \wedge \bigwedge_{i=1,2} (t_i \leq \tau_i) \wedge c(t_i)\big).$$

Proof. It is easy to see that $\models \varphi^* \to \varphi$. On the other hand, every model of φ can be turned into a model of φ^* by changing the extensions of t, t_1, t_2. ❑

Suppose X is a topological space, and S a regular closed subset of X. Then S is itself a topological space (with the subspace topology), which has its own regular closed algebra: $\boldsymbol{RC}(S) = \{S \cdot R \mid R \in \boldsymbol{RC}(X)\}$. Denoting the Boolean operations in $\boldsymbol{RC}(S)$ by $\cdot_S$ and $-_S$, etc., we have, for any $R_1, R_2 \in \boldsymbol{RC}(S)$: (i) $R_1 \cdot_S R_2 = R_1 \cdot R_2$; (ii) $-_S(R_1) = S \cdot (-R_1)$, (iii) $\mathbf{1}_S = S$ and $\mathbf{0}_S = \mathbf{0}$. For a formula φ and a variable s, define $\varphi_{|s}$ to be the result of replacing every maximal term τ occurring in φ by the term $s \cdot \tau$. For any model $\mathfrak{M} = (T, \cdot^{\mathfrak{M}})$, define $\mathfrak{M}_{|s}$ to be the model over the topological space $s^{\mathfrak{M}}$ (with the subspace topology) obtained by setting $r^{\mathfrak{M}_{|s}} = (r \cdot s)^{\mathfrak{M}}$ for all variables r.

Lemma 4. *For any $\mathcal{C}cc$-formula, $\mathfrak{M} \models \varphi_{|s}$ iff $\mathfrak{M}_{|s} \models \varphi$.*

Proof. One can show by induction that $(s \cdot \tau)^{\mathfrak{M}} = \tau^{\mathfrak{M}_{|s}}$, for any $\mathcal{B}$-term τ. ❑

Lemma 5. *Let $\varphi[C(\tau_1, \tau_2)]^-$ be a $\mathcal{C}cc$-formula, and s, t, t_1, t_2 fresh variables. Then φ is equi-satisfiable with the formula*

$$\varphi^* \;=\; \big(\varphi[t \neq \mathbf{0}]^-\big)_{|s} \;\wedge\; \big((t \cdot s = \mathbf{0}) \;\rightarrow\; \neg c(t_1 + t_2) \;\wedge \bigwedge_{i=1,2} c(t_i) \wedge (\tau_i \cdot s \leq t_i)\big).$$

Proof. Evidently, $\bigwedge_{i=1,2} (c(t_i) \wedge (\tau_i \cdot s \leq t_i)) \wedge \neg c(t_1 + t_2) \rightarrow \neg C(\tau_1 \cdot s, \tau_2 \cdot s)$ is a $\mathcal{C}cc$-validity. So any model $\mathfrak{A}$ of φ^* is a model of $(\varphi[C(\tau_1, \tau_2)]^-)_{|s}$, whence, by Lemma 4, $\mathfrak{A}_{|s} \models \varphi[C(\tau_1, \tau_2)]^-$. Conversely, suppose $\mathfrak{A} \models \varphi[C(\tau_1, \tau_2)]^-$, for a quasi-saw model $\mathfrak{A}$ induced by (W, R). Let W_i, $(i = 0, 1)$ be the set of points of depth i in (W, R). Without loss of generality, we may assume that every point in W_0 has an R-predecessor in W_1. If $\mathfrak{A} \models C(\tau_1, \tau_2)$, let $\mathfrak{A}^*$ be exactly like $\mathfrak{A}$ except that $s^{\mathfrak{A}^*}$ and $t^{\mathfrak{A}^*}$ are both the whole space. Then $\mathfrak{A}^* \models \varphi^*$. On the other hand, if $\mathfrak{A} \not\models C(\tau_1, \tau_2)$, we add, for $i = 1, 2$, an extra point u_i to W to connect up the points in $\tau_i^{\mathfrak{A}}$. Formally, let $W^* = W \cup \{u_1, u_2\}$, where $u_1, u_2 \notin W$, and let R^* be the reflexive closure of the union of R and $\{(z, u_i) \mid z \in \tau_i^{\mathfrak{A}} \cap W_1\}$, for $i = 1, 2$. Clearly, W is a regular closed subset of the topological space (W^*, R^*). Now define the interpretation $\mathfrak{A}^*$ over (W^*, R^*) by setting $s^{\mathfrak{A}^*} = W$, $t^{\mathfrak{A}^*} = \emptyset$, $t_i^{\mathfrak{A}^*} = \tau_i^{\mathfrak{A}} \cup \{u_i\}$ $(i = 1, 2)$, and $r^{\mathfrak{A}^*} = r^{\mathfrak{A}}$ for all other variables r. Thus, $\mathfrak{A} = \mathfrak{A}^*_{|s}$, whence, by Lemma 4, $\mathfrak{A}^* \models (\varphi[C(\tau_1, \tau_2)]^-)_{|s}$, and so $\mathfrak{A}^* \models (\varphi[t \neq \mathbf{0}]^-)_{|s}$. By construction, $\mathfrak{A}^* \models \bigwedge_{i=1,2} (c(t_i) \wedge (\tau_i \cdot s \leq t_i)) \wedge \neg c(t_1 + t_2)$. Thus, $\mathfrak{A}^* \models \varphi^*$. ❑

It follows from these lemmas that the satisfiability problem for $\mathcal{C}c$ (and $\mathcal{C}cc$) is reducible to the satisfiability problem for $\mathcal{B}c$ ($\mathcal{B}cc$, respectively). For, by repeated application of Lemmas 3 and 5, successive occurrences of C in a $\mathcal{C}c$- or $\mathcal{C}cc$-formula may be equisatisfiably eliminated, using only logarithmic space.

As a consequence, by Theorems 3 and 4, we obtain:

Theorem 5. *Satisfiability of $\mathcal{B}c$- and $\mathcal{B}cc$-formulas is, respectively,* ExpTime*- and* NExpTime*-complete.*

We remark in passing that the full reduction is not required for Theorem 5. For the proofs of Theorems 3 and 4 in fact rely on formulas in which conjuncts $C(\tau_1, \tau_2)$ occur only in negative contexts (and thus Lemma 5 is enough).

5 Discussion and Further Work

In this paper, we have reported on the computational complexity of the satisfiability problems for the spatial logics $\mathcal{B}$, $\mathcal{C}$ and $\mathcal{S}4_u$ extended with connectedness constraints. All these logics feature variables which range over subsets of topological spaces: regular subsets in the case of logics based on $\mathcal{B}$ and $\mathcal{C}$, and arbitrary subsets in the case of $\mathcal{S}4_u$. However, topological spaces form an extremely general category: and it is natural to ask what happens when we restrict consideration

to particular classes of topological spaces. Most saliently of all: what happens when these logics are interpreted over the *specific* topological spaces $\mathbb{R}^2$ or $\mathbb{R}^3$?

Without the ability to express connectedness, topological spatial logics are almost completely insensitive to the underlying topology. Thus, a $\mathcal{B}$-formula is satisfiable over $\mathbb{R}^n$, for any fixed n, iff it is satisfiable (over some space); a $\mathcal{C}$-formula is satisfiable over $\mathbb{R}^n$, for any fixed n, iff it is satisfiable over a connected space [23]; and an $\mathcal{S}4_u$-formula is satisfiable over $\mathbb{R}^n$, for any fixed n, iff it is satisfiable over a connected, dense-in-itself, separable metric space [21]. Adding connectedness constraints to these logics changes the situation radically, however. As a simple illustration, consider the $\mathcal{B}c$ formula

$$\bigwedge_{1\le i\le 3} c(r_i) \quad \wedge \quad \bigwedge_{1\le i<j\le 3} \big(r_i \cdot r_j \neq \mathbf{0}\big) \quad \wedge \quad \big(r_1 \cdot r_2 \cdot r_3 = \mathbf{0}\big),$$

which states that there are three pairwise overlapping, connected regions whose common part has an empty interior. Since connected subsets of $\mathbb{R}$ are intervals, this formula is not satisfiable over $\mathbb{R}$; yet it is satisfiable over $\mathbb{R}^n$, for any $n > 1$. Or again, it can be shown (see [14], p. 137) that the $\mathcal{S}4_u c$-formula

$$(v_1 \cap v_2 = \mathbf{0}) \ \wedge \ \bigwedge_{i=1,2} ((v_i^- \subseteq v_i) \ \wedge \ c(\overline{v_i})) \quad \wedge \quad \neg c(\overline{v_1} \cap \overline{v_2})$$

is not satisfiable over $\mathbb{R}^n$ (for any n); yet it is easily seen to be satisfiable over other manifolds (even of dimension 1!).

What can we say about the complexity of determining satisfiability over these spaces? In the one-dimensional case, matching complexity bounds are available.

Theorem 6. *Satisfiability of $\mathcal{S}4_u cc$-formulas in topological models based on $\mathbb{R}$ is* PSPACE*-complete.*

Proof. The proof is by reduction to the propositional temporal logic of the real line, for which satisfiability is known to be PSPACE-complete [19]. Since, for $\mathcal{C}$-formulas, satisfiability over connected spaces implies satisfiability over $\mathbb{R}$, it follows from [23] that this bound is tight. ❑

For $n > 1$, the work reported here yields lower-bound information for satisfiability over $\mathbb{R}^n$:

Theorem 7. *Satisfiability of $\mathcal{C}c$- and $\mathcal{C}cc$-formulas in topological models based on $\mathbb{R}^n$, for each $n > 1$, is* EXPTIME*- and* NEXPTIME*-hard, respectively.*

Proof. Based on the fact that the models constructed in the proofs of Theorem 3 and 4 can be turned into models over $\mathbb{R}^2$, and so over any $\mathbb{R}^n$, for $n \geq 2$. ❑

It follows that the EXPTIME and NEXPTIME lower bounds hold for satisfiability of $\mathcal{S}4_u c$- and $\mathcal{S}4_u cc$-formulas over $\mathbb{R}^n$, respectively.

We mention that, when variables are restricted to range over closed disc-homeomorphs in $\mathbb{R}^2$, then the problem of determining the satisfiability of $\mathcal{RCC}$-8-constraints is known to be in NP [20]—a very surprising result, since the smallest

satisfying drawings may involve exponentially many intersection points [11]. At present, no upper complexity bounds for the logics $\mathcal{B}c$, $\mathcal{B}cc$, $\mathcal{C}c$, $\mathcal{C}cc$, $\mathcal{S}4_u c$, interpreted over Euclidean spaces of fixed dimension greater than 1 are known.

Acknowledgements. The work on this paper was partially supported by the U.K. EPSRC research grants EP/E034942/1 and EP/E035248/1. We are grateful to Dimiter Vakarelov for comments and discussions.

References

1. Borgo, S., Guarino, N., Masolo, C.: A pointless theory of space based on strong connection and congruence. In: Aiello, L., Doyle, J., Shapiro, S. (eds.) KR, pp. 220–229. Morgan Kaufmann, San Francisco (1996)
2. Bourbaki, N.: General Topology, Part 1. Addison-Wesley, Hermann (1966)
3. Cantone, D., Cutello, V.: Decision algorithms for elementary topology I. Topological syllogistics with set and map constructs, connectedness and cardinailty composition. Comm. on Pure and Appl. Mathematics XLVII, 1197–1217 (1994)
4. Cohn, A., Renz, J.: Qualitative spatial representation and reasoning. In: van Hermelen, F., Lifschitz, V., Porter, B. (eds.) Handbook of Knowledge Representation, pp. 551–596. Elsevier, Amsterdam (2008)
5. Davis, E.: The expressivity of quantifying over regions. Journal of Logic and Computation 16, 891–916 (2006)
6. Dimov, G., Vakarelov, D.: Contact algebras and region-based theory of space: A proximity approach, I. Fundamenta Informaticae 74, 209–249 (2006)
7. Dornheim, C.: Undecidability of plane polygonal mereotopology. In: Cohn, A., Schubert, L., Shapiro, S. (eds.) KR, pp. 342–353. Morgan Kaufmann, San Francisco (1998)
8. Egenhofer, M., Franzosa, R.: Point-set topological spatial relations. International Journal of Geographical Information Systems 5, 161–174 (1991)
9. De Giacomo, G.: Decidability of Class-Based Knowledge Representation Formalisms. PhD thesis, Università degli Studi di Roma 'La Sapienza' (1995)
10. Grzegorczyk, A.: Undecidability of some topological theories. Fundamenta Mathematicae 38, 137–152 (1951)
11. Kratochvíl, J., Matoušek, J.: String graphs requiring exponential representations. J. of Combinatorial Theory, Series B 53, 1–4 (1991)
12. Lutz, C., Wolter, F.: Modal logics of topological relations. Logical Methods in Computer Science, 2 (2006)
13. McKinsey, J.C.C., Tarski, A.: The algebra of topology. Annals of Mathematics 45, 141–191 (1944)
14. Newman, M.: Elements of the Topology of Plane Sets of Points. Cambridge (1964)
15. Pratt-Hartmann, I.: A topological constraint language with component counting. Journal of Applied Non-Classical Logics 12, 441–467 (2002)
16. Randell, D., Cui, Z., Cohn, A.: A spatial logic based on regions and connection. In: Nebel, B., Rich, C., Swartout, W. (eds.) Proceedings of KR, pp. 165–176. Morgan Kaufmann, San Francisco (1992)
17. Renz, J.: A canonical model of the region connection calculus. In: Cohn, A., Schubert, L., Shapiro, S. (eds.) KR, pp. 330–341. Morgan Kaufmann, San Francisco (1998)

18. Renz, J., Nebel, B.: Qualitative spatial reasoning using constraint calculi. In: Aiello, M., Pratt-Hartmann, I., van Benthem, J. (eds.) Handbook of Spatial Logics, pp. 161–216. Springer, Heidelberg (2007)
19. Reynolds, M.: The complexity of the temporal logic over the reals (Manuscript, 2008), http://www.csse.uwa.edu.au/~mark/research/Online/CORT.htm
20. Schaefer, M., Sedgwick, E., Štefankovič, D.: Recognizing string graphs in NP. Journal of Computer and System Sciences 67, 365–380 (2003)
21. Shehtman, V.: Everywhere and Here. Journal of Applied Non-Classical Logics 9, 369–380 (1999)
22. Whitehead, A.N.: Process and Reality. MacMillan Company, New York (1929)
23. Wolter, F., Zakharyaschev, M.: Spatial reasoning in RCC-8 with Boolean region terms. In: Horn, W. (ed.) Proceedings of ECAI, pp. 244–248. IOS Press, Amsterdam (2000)

Decidable and Undecidable Fragments of Halpern and Shoham's Interval Temporal Logic: Towards a Complete Classification

Davide Bresolin[1], Dario Della Monica[2], Valentin Goranko[3], Angelo Montanari[2], and Guido Sciavicco[4,⋆]

[1] University of Verona, Verona (Italy)
davide.bresolin@univr.it
[2] University of Udine, Udine (Italy)
{dario.dellamonica,angelo.montanari}@dimi.uniud.it
[3] University of Witwatersrand, Johannesburg (South Africa)
valentin.goranko@wits.ac.za
[4] University of Murcia, Murcia (Spain)
guido@um.es

Abstract. Interval temporal logics are based on temporal structures where time intervals, rather than time instants, are the primitive ontological entities. They employ modal operators corresponding to various relations between intervals, known as Allen's relations. Technically, validity in interval temporal logics translates to dyadic second-order logic, thus explaining their complex computational behavior. The full modal logic of Allen's relations, called HS, has been proved to be undecidable by Halpern and Shoham under very weak assumptions on the class of interval structures, and this result was discouraging attempts for practical applications and further research in the field. A renewed interest has been recently stimulated by the discovery of interesting decidable fragments of HS. This paper contributes to the characterization of the boundary between decidability and undecidability of HS fragments. It summarizes known positive and negative results, it describes the main techniques applied so far in both directions, and it establishes a number of new undecidability results for relatively small fragments of HS.

1 Introduction

Interval temporal logics are based on interval structures over linearly ordered domains, where time intervals, rather than time instants, are the primitive ontological entities. The variety of relations between intervals in linear orders was first studied systematically by Allen [1], who explored their use in systems for time management and planning. Interval reasoning arises naturally in various other fields of artificial intelligence, such as theories of action and change, natural language analysis and processing, and constraint satisfaction problems. Temporal

⋆ Guido Sciavicco was co-financed by the Spanish projects TIN 2006-15460-C04-01 and PET 2006_0406.

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 590–604, 2008.

logics with interval-based semantics have also been proposed as a useful formalism for the specification and verification of hardware [21] and of real-time systems [11]. Thus, the relevance of interval temporal logics in many areas of artificial intelligence and computer science is nowadays widely recognized.

Interval temporal logics feature modal operators corresponding to various possible relations over intervals. A special role is played by the thirteen different binary relations (on linear orders) known as Allen's relations. In [15], Halpern and Shoham introduce a modal logic for reasoning about interval structures, called HS, with modal operators corresponding to Allen's interval relations. Formulas of HS are evaluated at intervals, i.e., pairs of points, and, consequently, they translate into binary relations in interval models. Accordingly, validity in HS translates to dyadic second-order logic, thus causing its complex and generally bad computational behavior, where undecidability is the common case and decidability is usually achieved by imposing severe restrictions on the interval-based semantics, which essentially reduce it to a point-based one. More precisely, HS turns out to be undecidable under very weak assumptions on the class of interval structures [15]: we get undecidability for any class of interval structures over linear orders that contains at least one linear order with an infinite ascending (or descending) chain, thus including all natural numerical time-flows $\mathbb{N}, \mathbb{Z}, \mathbb{Q}$, and $\mathbb{R}$.

For a long time, such a sweeping undecidability result has discouraged attempts for practical applications and further research on interval logics. A renewed interest in the area has been recently stimulated by the discovery of some interesting decidable fragments of HS [3,4,5,6,7,9]. As an effect, the identification of expressive enough decidable fragments of HS has been added to the current research agenda for (interval) temporal logic. While the algebra of Allen's relations, the so-called Allen's Interval Algebra, has been extensively studied and completely classified from the point of view of computational complexity [17] (tractability/intractability of the consistency problem for fragments of Interval Algebra), the characterization of decidable/undecidable fragments of the modal logic of Allen's relations (HS) is considerably harder.

This paper aims at contributing to the identification of the boundary between decidability and undecidability of HS fragments. It summarizes known positive and negative results, it presents the main techniques so far exploited in both directions, and it establishes new undecidability results. Two important parameters of the proposed classification are the set of modalities of the fragment and the class of linearly ordered sets in which it is interpreted. We shall take into consideration the full set of modal operators corresponding to Allen's relations as defined in HS, apart for the trivial one corresponding to equality, plus two definable modalities, namely, those for the *proper during* relation and its inverse *proper contains* (the interval logic of the *proper during* relation has been recently shown to be decidable on dense orders [3]).

The paper is structured as follows. In the next section, we introduce the framework of interval-based temporal logic with unary modalities. In Section 3, we give an up-to-date survey of known decidable fragments. In Section 4, we first summarize known undecidability results and then we provide a number of new

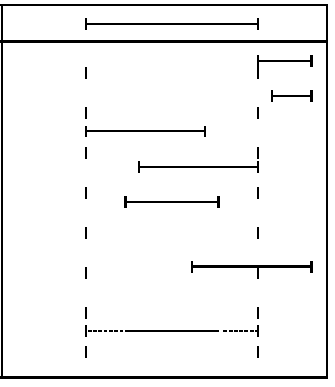
Op.	Semantics	
$\langle A \rangle$	$\mathbf{M}, [a,b] \Vdash \langle A \rangle \phi \Leftrightarrow \exists c (b < c . \mathbf{M}, [b,c] \Vdash \phi)$	
$\langle L \rangle$	$\mathbf{M}, [a,b] \Vdash \langle L \rangle \phi \Leftrightarrow \exists c,d (b < c < d . \mathbf{M}, [c,d] \Vdash \phi)$	
$\langle B \rangle$	$\mathbf{M}, [a,b] \Vdash \langle B \rangle \phi \Leftrightarrow \exists c (a \leq c < b . \mathbf{M}, [a,c] \Vdash \phi)$	
$\langle E \rangle$	$\mathbf{M}, [a,b] \Vdash \langle E \rangle \phi \Leftrightarrow \exists c (a < c \leq b . \mathbf{M}, [c,b] \Vdash \phi)$	
$\langle D \rangle$	$\mathbf{M}, [a,b] \Vdash \langle D \rangle \phi \Leftrightarrow \exists c,d (a < c \leq d < b . \mathbf{M}, [c,d] \Vdash \phi)$	
$\langle O \rangle$	$\mathbf{M}, [a,b] \Vdash \langle O \rangle \phi \Leftrightarrow \exists c,d (a < c \leq b < d . \mathbf{M}, [c,d] \Vdash \phi)$	
$\langle D \rangle_{\sqsubset}$	$\mathbf{M}, [a,b] \Vdash \langle D \rangle_{\sqsubset} \phi \Leftrightarrow \exists c,d (a \leq c \leq d \leq b . \mathbf{M}, [c,d] \Vdash \phi \wedge [c,d] \neq [a,b])$	

Fig. 1. Formal semantics for some interval operators

undecidability results for other fragments of HS by reduction from the octant and the $\mathbb{N} \times \mathbb{N}$ tiling problems.

2 Interval Logics over Linearly Ordered Sets

Let $\mathbb{D} = \langle D, < \rangle$ be a linearly ordered set. An *interval* over $\mathbb{D}$ is an ordered pair $[a,b]$, where $a, b \in D$ and $a \leq b$. Intervals of the type $[a,a]$ are called *point intervals*; if these are excluded, the resulting semantics is called *strict interval semantics* (*non-strict* otherwise). In this paper, we take the more standard non-strict semantics as default. The language of a propositional interval logic consists of a set $\mathcal{AP}$ of propositional letters, any complete set of classical operators (such as $\vee$ and $\neg$), and a set of modal operators $\langle X_1 \rangle, \ldots, \langle X_k \rangle$, each of them associated with a specific binary relation over intervals[1]. Formulas are defined by the following grammar:

$$\varphi ::= p \mid \pi \mid \neg\varphi \mid \varphi \vee \varphi \mid \langle X_1 \rangle \varphi \mid \ldots \mid \langle X_k \rangle \varphi,$$

where π is a modal constant, true precisely at point intervals. We omit π when it is definable in the language or when the strict semantics is adopted.

The semantics of an interval-based temporal logic is given in terms of *interval models* $\mathbf{M} = \langle \mathbb{I}(\mathbb{D}), V \rangle$, where $\mathbb{I}(\mathbb{D})$ is the set of all intervals over $\mathbb{D}$ and the *valuation function* $V : \mathcal{AP} \mapsto 2^{\mathbb{I}(\mathbb{D})}$ assigns to every $p \in \mathcal{AP}$ the set of intervals $V(p)$ over which it holds. The *truth of a formula over a given interval* $[a,b]$ in a model $\mathbf{M}$ is defined by structural induction on formulas:

- $\mathbf{M}, [a,b] \Vdash \pi$ iff $a = b$;
- $\mathbf{M}, [a,b] \Vdash p$ iff $[a,b] \in V(p)$, for all $p \in \mathcal{AP}$;
- $\mathbf{M}, [a,b] \Vdash \neg\psi$ iff it is not the case that $\mathbf{M}, [a,b] \Vdash \psi$;

[1] In this paper, we restrict our attention to *unary* modal operators only (decidability issues for *binary* modal operators are addressed in [16]).

- $\mathbf{M}, [a,b] \Vdash \varphi \vee \psi$ iff $\mathbf{M}, [a,b] \Vdash \varphi$ or $\mathbf{M}, [a,b] \Vdash \psi$;
- $\mathbf{M}, [a,b] \Vdash \langle X_i \rangle \psi$ iff there exists an interval $[c,d]$ such that $[a,b]\ R_{X_i}\ [c,d]$, and $\mathbf{M}, [c,d] \Vdash \psi$,

where R_{X_i} is the (binary) interval relation corresponding to the modal operator $\langle X_i \rangle$. In Figure 1 we list the most common unary interval operators and their semantics. Moreover, we denote by $\langle \overline{X} \rangle$ the transpose of each modal operator $\langle X \rangle$, which corresponds to the inverse of the relation R_X. Except for *proper during* and its inverse [3], these are precisely Allen's interval relations [1]. It is easy to show that some of these modal operators are definable in terms of others (some of these definitions do not work with the strict semantics), e.g., $\langle D \rangle p = \langle B \rangle \langle E \rangle p$, $\langle D \rangle_{\sqsubset} p = \langle B \rangle p \vee \langle E \rangle p \vee \langle B \rangle \langle E \rangle p$, $\langle A \rangle p = \langle E \rangle ([B] \bot \wedge \langle \overline{B} \rangle p) \vee ([E] \bot \wedge \langle \overline{B} \rangle p)$, $\langle L \rangle p = \langle A \rangle \langle A \rangle p$, $\langle O \rangle p = \langle E \rangle \langle \overline{B} \rangle p$, and likewise for their transposes. Moreover, the modal constant π is definable in most sufficiently rich languages, viz.:

$$\pi = [B] \bot = [E] \bot = [O] \bot = [\overline{O}] \bot = [D]_{\sqsubset} \bot. \tag{1}$$

Thus, eventually, all operators corresponding to Allen's interval relations turn out to be definable in terms of $\langle B \rangle$,$\langle E \rangle$, and their transposes (as a matter of fact, $\langle A \rangle$ was included in the original formulation of HS; its definability in terms of the other operators was later shown in [23]).

Here we will consider all HS fragments and for that purpose we will assume all operators listed in Figure 1 (and their transposes) to be primitive in the language. In general, when referring to a specific fragment of HS, we name it by its modal operators. For example, the fragment featuring the operators $\langle B \rangle, \langle E \rangle$ will be denoted by BE.

Besides the usual $\mathbb{N}, \mathbb{Z}$, and $\mathbb{Q}$, we introduce a suitable notation for some common classes of strict linear orders:

- Lin = the class of all linear orders;
- Fin = the class of all **finite** linear orders;
- Den = the class of all **dense** linear orders;
- Dis = the class of all **discrete** linear orders;
- Asc = the class of all linear orders **with an infinite ascending sequence**;
- Des = the class of all linear orders **with an infinite descending sequence**.

3 Decidable Fragments of HS

In this section, we briefly survey the maximal known decidable fragments of HS.

All early decidability results about interval logics were based on severe restrictions of the interval-based semantics, essentially reducing it to a point-based one. Such restrictions include *locality*, according to which all atomic propositions are point-wise and truth over an interval is defined as truth at its initial point, and *homogeneity*, according to which truth of a formula over an interval implies truth of that formula over every sub-interval. By imposing such constraints, decidability of HS can be proved by embedding it into linear temporal logic [21,23]. Decidability can also be achieved by constraining the class of temporal structures over

which the logic is interpreted. This is the case with *split-structures*, where any interval can be "chopped" in at most one way. The decidability of various interval logics, including HS, interpreted over split-structures, has been proved by embedding them into first-order decidable theories of time granularity [20].

For some simple fragments of HS, like $\mathsf{B}\overline{\mathsf{B}}$ and $\mathsf{E}\overline{\mathsf{E}}$, decidability has been obtained without any semantic restriction by means of direct translation to the point-based semantics and reduction to decidability of respective point-based temporal logics [14]. In any of these logics, one of the endpoints of every interval related to the current one remains fixed, thereby reducing the interval-based semantics to the point-based one by mapping every interval of the generated submodel to its non-fixed endpoint. Consequently, these fragments can be polynomially translated to the linear time Temporal Logic with Future and Past TL[F,P], thus proving that they are NP-complete when interpreted on the class of all linearly ordered sets or on any of $\mathbb{N}$, $\mathbb{Q}$, and $\mathbb{R}$ [12,14].

Decidability results for fragments of HS with unrestricted interval-based semantics, non-reducible to point-based one, have been recently obtained by means of a translation method. This is the case with $\mathsf{A}\overline{\mathsf{A}}$, also known as *Propositional Neighborhood Logic* (PNL) [13][2]. In [6,7], decidability in NEXPTIME of $\mathsf{A}\overline{\mathsf{A}}$ has been proved by translation to the two-variable fragment of first-order logic with binary relations over linear domains $\mathrm{FO}^2[<]$ and reference to the NEXPTIME-complete decidability result for $\mathrm{FO}^2[<]$ by Otto [22] (for proof details and NEXPTIME-hardness, we refer the reader to [6,7]). Otto's results, and consequently the decidability of $\mathsf{A}\overline{\mathsf{A}}$, apply not only to the class of all linear orders, but also to some natural subclasses of it, such as the class of all well-founded linear orders, the class of all finite linear orders, and $\mathbb{N}$.

Finally, decidability of some fragments of HS has been demonstrated by taking advantage of the small model property with respect to suitable classes of satisfiability preserving *pseudo-models*. This method has been successfully applied to the logics of subintervals D and $\mathsf{D}_{\sqsubset}$, interpreted over dense linear orders [3,4,5], and to the logic $\mathsf{A}\overline{\mathsf{A}}$ (resp., A), interpreted over $\mathbb{Z}$ (resp., $\mathbb{N}$) [8,10]. In [3,4,5], Bresolin et al. make use of this technique to develop optimal tableau systems for D and $\mathsf{D}_{\sqsubset}$ that work in PSPACE. (NEXPTIME) tableau-based decision procedures for $\mathsf{A}\overline{\mathsf{A}}$ over $\mathbb{Z}$ and A over $\mathbb{N}$ have been developed in [8,10]. The tableau system for A over $\mathbb{N}$ has been recently generalized to the case of all linearly ordered domains [9].

4 Undecidable Fragments of HS

Undecidable fragments of HS are much more common than decidable ones. In the following, we first summarize some well-known undecidability results, which have been proved by means of a reduction from the non-halting problem for Turing Machines. Then, we recall recent undecidability results for 6 fragments of HS that properly extend $\mathsf{A}\overline{\mathsf{A}}$, namely, $\mathsf{A}\overline{\mathsf{A}}\mathsf{B}\overline{\mathsf{E}}$, $\mathsf{A}\overline{\mathsf{A}}\mathsf{E}\overline{\mathsf{B}}$, and $\mathsf{A}\overline{\mathsf{A}}\mathsf{D}^*$, where $\mathsf{D}^* \in \{\mathsf{D}, \overline{\mathsf{D}}, \mathsf{D}_{\sqsubset}, \overline{\mathsf{D}}_{\sqsubset}\}$, interpreted over any class of linear orders containing a linear order

[2] Since L and $\overline{\mathsf{L}}$ are definable in $\mathsf{A}\overline{\mathsf{A}}$, decidability of this fragment actually implies decidability of $\mathsf{A}\overline{\mathsf{A}}\mathsf{L}\overline{\mathsf{L}}$.

with an infinite chain, which have been obtained by means of an encoding from the octant tiling problem [7]. Next, we show that a similar reduction from the octant tiling problem can be exploited to prove the undecidability of other 24 fragments of HS, namely, $\mathsf{AD}^*\mathsf{E}$, $\mathsf{AD}^*\overline{\mathsf{E}}$, and $\mathsf{AD}^*\overline{\mathsf{O}}$ (over any class of linear orders containing a linear order with an infinite ascending chain), $\overline{\mathsf{A}}\mathsf{D}^*\mathsf{B}$, $\overline{\mathsf{A}}\mathsf{D}^*\overline{\mathsf{B}}$, and $\overline{\mathsf{A}}\mathsf{D}^*\mathsf{O}$ (over any class of linear orders containing a linear order with an infinite descending chain). Finally, we take advantage of a reduction from the $\mathbb{N} \times \mathbb{N}$ tiling problem to prove the undecidability of $\mathsf{B}\overline{\mathsf{E}}$, $\overline{\mathsf{B}}\mathsf{E}$, and $\overline{\mathsf{B}\mathsf{E}}$ over the appropriate classes of linear orders, thus improving the results for $\mathsf{A}\overline{\mathsf{A}}\mathsf{B}\overline{\mathsf{E}}$ and $\mathsf{A}\overline{\mathsf{A}}\mathsf{E}\overline{\mathsf{B}}$ given in [7].

4.1 Reduction from the Non-halting Problem

The undecidability of HS with respect to most classes of linear orders has been proved by means of a reduction from the non-halting problem for Turing Machines [15] (in fact, the reduction is to any of the fragments ABE and $\overline{\mathsf{A}}\mathsf{BE}$).

Theorem 1 (Halpern and Shoham [15]). *The satisfiability problem for* ABE *is undecidable in any class of linear orders that contains at least one linear order with an infinite ascending sequence (in particular, in* Lin, Den, Dis, $\mathbb{Z}, \mathbb{Q}, \mathbb{R}$, Asc, *and* $\mathbb{N}$*). Similarly, the satisfiability problem for* $\overline{\mathsf{A}}\mathsf{BE}$ *is undecidable in each of the classes* Lin, Den, Dis, $\mathbb{Z}, \mathbb{Q}, \mathbb{R}$, Des, *and* $\mathbb{Z}^-$.

The undecidability of the satisfiability problem for HS in all the classes Theorem 1 refers to immediately follows.

In [18], Lodaya shows that a suitable sharpening of the reduction technique from [15] can be exploited to prove the undecidability of the fragment BE over dense linear orders (thus strengthening Halpern and Shoham's result in this restricted setting). As a preliminary result, he proves that the logic with the binary *chop* operator C, that splits an interval in two parts (and is not definable in HS), and the modal constant π is undecidable by means of an adaptation of the proof for HS. Then, he shows that the operators $\langle B \rangle$ and $\langle E \rangle$, which can be easily defined in terms of C and π, suffice for undecidability. In [14] it was observed that this result actually applies to the class of all linear orders.

Theorem 2 (Lodaya, Goranko et al. [14,18]). *The satisfiability problem for* BE *is undecidable in the classes* Lin *and* Den.

4.2 Reduction from the Octant Tiling Problem

The undecidability of a number of HS fragments has been proved by using variations of a reduction from the *unbounded tiling problem* for the second octant $\mathcal{O}$ of the integer plane. This is the problem of establishing whether a given finite set of tile types $\mathcal{T} = \{t_1, \ldots, t_k\}$ can tile $\mathcal{O} = \{(i, j) : i, j \in \mathbb{N} \wedge 0 \leq i \leq j\}$. This problem can be shown to be undecidable by a simple application of the König's Lemma in the same way as it was used in [2] to show the undecidability of the $\mathbb{N} \times \mathbb{N}$ tiling problem from that of $\mathbb{Z} \times \mathbb{Z}$ one. For every tile type $t_i \in \mathcal{T}$, let $right(t_i)$, $left(t_i)$,

$up(t_i)$, and $down(t_i)$ be the colors of the corresponding sides of t_i. To solve the problem, one must find a function $f : \mathcal{O} \to \mathcal{T}$ such that

$$right(f(n,m)) = left(f(n+1,m))$$

and

$$up(f(n,m)) = down(f(n,m+1)).$$

In [7], a reduction from the unbounded tiling problem for the second octant $\mathcal{O}$ of the integer plane has been applied to prove the undecidability of the extensions of $\mathsf{A\overline{A}}$ with any of the operators $\langle D\rangle$, $\langle \overline{D}\rangle$, $\langle D\rangle_{\sqsubset}$, and $\langle \overline{D}\rangle_{\sqsubset}$, or with the pairs of operators $\langle B\rangle\langle \overline{E}\rangle$ or $\langle \overline{B}\rangle\langle E\rangle$, interpreted in any class of linear orders containing a linear order with an infinite (ascending or descending) chain.

Theorem 3 (Bresolin et al. [7]). *The satisfiability problem for each of the fragments* $\mathsf{A\overline{A}D^*}$, $\mathsf{A\overline{A}B\overline{E}}$, *and* $\mathsf{A\overline{A}E\overline{B}}$ *is undecidable in each of the classes* Lin, Den, Dis, $\mathbb{Z}, \mathbb{Q}, \mathbb{R}$, Des, Asc, $\mathbb{N}$, *and* $\mathbb{Z}^-$.

In the following, we will show that similar reductions can be exploited to prove the undecidability of other meaningful fragments of HS.

Theorem 4. *The satisfiability problem for each of the fragments* $\mathsf{AD^*E}$, $\mathsf{AD^*\overline{E}}$, *and* $\mathsf{AD^*\overline{O}}$ *is undecidable in any class of linear orders containing a linear order with an infinite ascending chain. Likewise, the satisfiability problem for the fragments* $\mathsf{\overline{A}D^*B}$, $\mathsf{\overline{A}D^*\overline{B}}$, *and* $\mathsf{\overline{A}D^*O}$ *is undecidable in any class of linear orders containing a linear order with an infinite descending chain.*

We give the details of the proof for the case ADE; the other cases are quite similar. We consider a signature containing, inter alia, the special propositional letters u, tile, Id, $\mathtt{t_1}, \ldots, \mathtt{t_k}$, bb, be, eb, and corr.

Unit-intervals. We set our framework by forcing the existence of a unique infinite chain of so-called *unit-intervals* (for short, u-*intervals*) on the linear order, which covers an initial segment of the model. These u-intervals will be labeled by the propositional variable u. They will be used as cells to arrange the tiling. First of all, we define an *always in the future* modality which captures future intervals only:

$$[G]p = p \wedge [A]p \wedge [A][A]p.$$

Then, u-*intervals* can be encoded as follows:

$$B_1 = \neg\mathsf{u} \wedge \langle A\rangle\mathsf{u} \wedge [G](\mathsf{u} \to (\neg\pi \wedge \langle A\rangle\mathsf{u} \wedge \neg\langle D\rangle\mathsf{u} \wedge \neg\langle D\rangle\langle A\rangle\mathsf{u})),$$
$$B_2 = [G]\textstyle\bigwedge_{p\in\mathcal{AP}}((p \vee \langle A\rangle p) \to \langle A\rangle\mathsf{u}).$$

Formula B_2 restricts our domain of 'legitimate intervals' to those composed of u-intervals, while B_1 guarantees the existence of an infinite sequence of consecutive u-intervals, thus implying the following lemma.

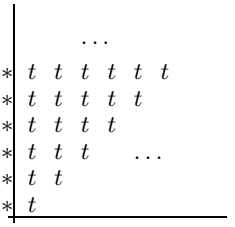


Fig. 2. A schema of the encoding (we abbreviate `tile` as t)

Lemma 1. *Suppose that* $\mathbf{M}, [a, b] \Vdash B_1$*. Then, there exists an infinite sequence of points* $b_0 < b_1 < \ldots$ *in* $\mathbf{M}$*, such that* $b_0 = b$*, for each* i*,* $\mathbf{M}, [b_i, b_{i+1}] \Vdash \mathsf{u}$*, and no other interval* $[c, d]$*, with* $c \neq d$*, in* $\mathbf{M}$ *satisfies* u*, unless* $c > b_i$ *for every* $i \in \mathbb{N}$*, or* $c < b$*.*

Encoding a tile. Every u-interval will represent either a tile or a special marker, denoted by $*$, that identifies the border between two ld-intervals (ld-intervals represent the rows of the tiling and will be defined later). Formally, we put:

$$B_3 = [G](\mathsf{u} \leftrightarrow (* \vee \mathsf{tile})) \wedge [G](* \rightarrow \neg \mathsf{tile}) \wedge [G]\neg(* \wedge \langle A \rangle *),$$
$$B_4 = [G](\mathsf{tile} \leftrightarrow (\bigvee_{i=1}^{k} \mathtt{t_i} \wedge \bigwedge_{i,j=1, i \neq j}^{k} \neg(\mathtt{t_i} \wedge \mathtt{t_j}))).$$

If a tile is placed on a u-interval $[a, b]$, we call a and b respectively the *beginning point* and the *ending point* of that tile.

Encoding rows of the tiling. An ld-interval (or just ld) is an interval consisting of a finite sequence of at least two u-subintervals. Each ld represents a row (level) of the tiling of $\mathcal{O}$. The first u-subinterval in an ld is a $*$-interval and every following u-subinterval is the encoding of a tile (see Figure 2). The ld-intervals representing the bottom-up consecutive levels of the tiling of $\mathcal{O}$ are arranged one after another in a chain. The first ld is composed by a single tile. To prevent the existence of interleaving sequences of ld-intervals, we do not allow occurrences of $*$-subintervals inside an ld. These conditions are imposed by the following formulas:

$$B_5 = [G]((\mathsf{ld} \rightarrow (\neg \mathsf{u} \wedge \langle A \rangle \mathsf{ld} \wedge \neg \langle D \rangle \langle A \rangle \mathsf{ld}))) \wedge [G](\langle A \rangle \mathsf{ld} \leftrightarrow \langle A \rangle *),$$
$$B_6 = \langle A \rangle (* \wedge \langle A \rangle (\mathsf{tile} \wedge \langle A \rangle *)),$$
$$B_7 = B_1 \wedge B_2 \wedge B_3 \wedge B_4 \wedge B_5 \wedge B_6.$$

Lemma 2. *Let* $\mathbf{M}, [a, b] \Vdash B_7$*. Then, there is a sequence of points* $b = b_1^0 < b_1^1 < \ldots b_1^{k_1} = b_2^0 < b_2^1 < \ldots < b_2^{k_2} = b_3^0 < \ldots$*, such that* $k_1 = 2$ *and for every* j*:*

1. $\mathbf{M}, [b_j^0, b_j^{k_j}] \Vdash \mathsf{ld}$ *and no other interval* $[c, d]$*, with* $c \neq d$*, in* $\mathbf{M}$ *is an* ld*-interval, unless possibly for* $c > b_j^{k_j}$ *for every* $j \in \mathbb{N}$*, or* $c < b$*;*
2. $\mathbf{M}, [b_j^0, b_j^1] \Vdash *$ *and no other interval* $[c, d]$*, with* $c \neq d$*, in* $\mathbf{M}$ *is a* $*$*-interval, unless possibly for* $c > b_j^{k_j}$ *for every* $j \in \mathbb{N}$*, or* $c < b$*;*

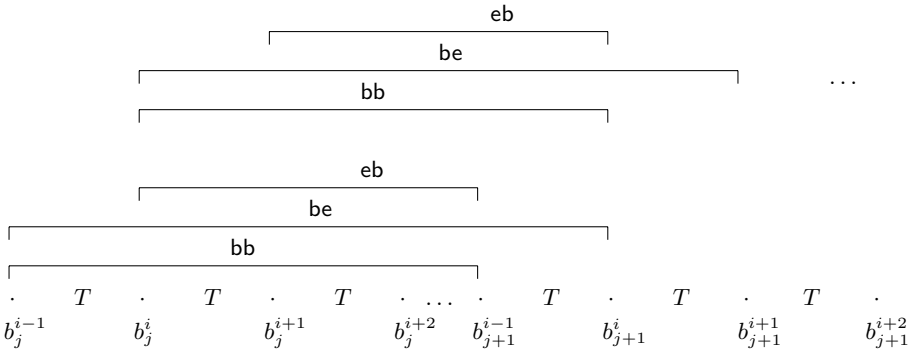


Fig. 3. A representation of bb, be, and eb-intervals

3. *for every* i *such that* $0 < i < k_j$, $\mathbf{M}, [b_j^i, b_j^{i+1}] \Vdash \mathsf{tile}$, *and no other interval* $[c, d]$, *with* $c \neq d$, *in* $\mathbf{M}$ *is a* tile*-interval, unless possibly for* $c > b_j^{k_j}$ *for every* $j \in \mathbb{N}$, *or* $c < b$.

Definition 1. *Let* $\mathbf{M}, [a, b] \Vdash B_7$ *and* $b_1^0 < b_1^1 < \ldots b_1^{k_1} = b_2^0 < b_2^1 < \ldots < b_2^{k_2} = b_3^0 \ldots$ *be the sequence of points whose existence is guaranteed by Lemma 2. For any* j, *the interval* $[b_j^0, b_j^{k_j}]$ *is the* j-th ld-interval of the sequence *and, for any* $i \geq 1$, *the interval* $[b_j^i, b_j^{i+1}]$ *is the* i-th tile of the ld-interval $[b_j^0, b_j^{k_j}]$.

Corresponding tiles. So far we have that, given a starting interval, the formula B_7 forces the underlying linearly ordered set to be, in the future of the current interval, a sequence of ld's, the first one of which containing exactly one tile. Now, we want to make sure that each tile at a certain *level* in $\mathcal{O}$ (i.e., ld) always has its corresponding tile at the immediate upper level. To this end, we will take advantage of some auxiliary propositional variables, namely, bb, which is to connect the beginning point of a tile to the beginning point of the corresponding tile above, be, which is to connect the beginning point of a tile to the ending point of the corresponding tile above, and eb, which is to connect the ending point of a tile to the beginning point of the corresponding tile above. If an interval is labeled with any of bb, eb, or be, we call it a *corresponding interval*, abbreviated corr*-interval*. A pictorial representation is given in Figure 3. The next formulas force corr-intervals to respect suitable properties so that all models satisfying them encode a correct tiling.

$$
\begin{aligned}
B_8 &= [G]((\mathsf{bb} \vee \mathsf{be} \vee \mathsf{eb}) \leftrightarrow \mathsf{corr}),\\
B_9 &= [G]\neg(\mathsf{corr} \wedge \mathsf{ld}),\\
B_{10} &= [G]((\mathsf{corr} \rightarrow \neg\langle D\rangle \mathsf{ld}) \wedge (\mathsf{ld} \rightarrow \neg\langle D\rangle \mathsf{corr})),\\
B_{11} &= [G]((\mathsf{corr} \rightarrow \neg\langle A\rangle \mathsf{ld}) \wedge (\langle A\rangle(\mathsf{bb} \vee \mathsf{be}) \rightarrow \neg\langle A\rangle \mathsf{ld})),\\
B_{12} &= B_8 \wedge B_9 \wedge B_{10} \wedge B_{11}.
\end{aligned}
$$

Lemma 3. *Let* $\mathbf{M}, [a, b] \Vdash B_7 \wedge B_{12}$. *Then, no* ld*-interval in* $\mathbf{M}$ *coincides with a* corr*-interval, nor is properly contained in a* corr*-interval, nor a* corr*-interval is properly contained in an* ld*-interval, unless it is an* eb*-interval beginning an* ld.

The next set of formulas guarantees that the corr-intervals satisfy the respective correspondences.

$B_{13} = [G](\langle A\rangle \mathsf{tile} \leftrightarrow \langle A\rangle \mathsf{bb})$,
$B_{14} = [A](\langle A\rangle(\mathsf{tile} \wedge \langle A\rangle \mathsf{tile}) \leftrightarrow \langle E\rangle \mathsf{bb})$,
$B_{15} = [G](\langle A\rangle \mathsf{tile} \leftrightarrow \langle A\rangle \mathsf{be})$,
$B_{16} = [A](\langle E\rangle \mathsf{tile} \wedge \langle A\rangle \mathsf{tile}) \leftrightarrow \langle E\rangle \mathsf{be})$,
$B_{17} = [G](\mathsf{u} \rightarrow (\mathsf{tile} \leftrightarrow \langle A\rangle \mathsf{eb}))$,
$B_{18} = [A](\langle A\rangle(\mathsf{tile} \wedge \langle A\rangle \mathsf{tile}) \leftrightarrow \langle E\rangle \mathsf{eb})$,
$B_{19} = B_{13} \wedge B_{14} \wedge B_{15} \wedge B_{16} \wedge B_{17} \wedge B_{18}$.

Lemma 4. *Let* $\mathbf{M}, [a,b] \Vdash B_7 \wedge B_{12} \wedge B_{19}$ *and let* $b_1^0 < b_1^1 < b_1^2 = b_2^0 < b_2^1 < \ldots < b_2^{k_2} = b_3^0 < \ldots$ *be the sequence of points whose existence is guaranteed by Lemma 2. Then, for every* $i \geq 0, j \geq 1$*:*

1. b_j^i *is the beginning point of a* bb *and a* be *iff* $1 \leq i \leq k_j - 1$.
2. b_j^i *is the beginning point of an* eb *iff* $2 \leq i \leq k_j$.
3. b_j^i *is the ending point of a* bb *and an* eb *iff* $1 \leq i \leq k_j - 2$.
4. b_j^i *is the ending point of a* be *iff* $2 \leq i \leq k_j - 1$.

Definition 2. *Given two* tile*-intervals* $[c,d]$ *and* $[e,f]$ *in a model* $\mathbf{M}$*, we say that* $[c,d]$ corresponds to $[e,f]$ *if* $\mathbf{M}, [c,e] \Vdash \mathsf{bb}$ *and* $\mathbf{M}, [c,f] \Vdash \mathsf{be}$ *and* $\mathbf{M}, [d,e] \Vdash \mathsf{eb}$.

The following formulas state the basic relationships between the three types of correspondence:

$B_{20} = [G] \bigwedge_{c,c' \in \{\mathsf{bb},\mathsf{eb},\mathsf{be}\}, c \neq c'} \neg(c \wedge c')$,
$B_{21} = [G](\mathsf{bb} \rightarrow \neg\langle D\rangle \mathsf{bb} \wedge \neg\langle D\rangle \mathsf{eb} \wedge \neg\langle D\rangle \mathsf{be})$,
$B_{22} = [G](\mathsf{eb} \rightarrow \neg\langle D\rangle \mathsf{bb} \wedge \neg\langle D\rangle \mathsf{eb} \wedge \neg\langle D\rangle \mathsf{be})$,
$B_{23} = [G](\mathsf{be} \rightarrow \langle D\rangle \mathsf{eb} \wedge \neg\langle D\rangle \mathsf{bb} \wedge \neg\langle D\rangle \mathsf{be})$,
$B_{24} = B_{20} \wedge B_{21} \wedge B_{22} \wedge B_{23}$.

Lemma 5. *Let* $\mathbf{M}, [a,b] \Vdash B_7 \wedge B_{12} \wedge B_{19} \wedge B_{24}$*. Then, for any* $j \geq 1$ *and* $i \geq 1$*:*

1. *the* i*-th tile of the* j*-th* ld*-interval corresponds to the* i*-th tile of the* $j+1$*-th* ld*-interval.*
2. *there are exactly* j *tiles in the* j*-th* ld*-interval.*
3. *no tile of the* j*-th* ld*-interval corresponds to the last tile of the* $j+1$*-th* ld*-interval.*

Encoding the tiling problem. We are now ready to show how to encode the octant tiling problem. Let $\phi_{\mathcal{T}}$ be the conjunction of B_7, B_{12}, B_{19}, B_{24}, B_{25}, and B_{26}, where B_{25} and B_{26} are the following two formulas:

$B_{25} = [G]((\mathsf{tile} \wedge \langle A\rangle \mathsf{tile}) \rightarrow \bigvee_{right(t_i) = left(t_j)} (\mathtt{t_i} \wedge \langle A\rangle \mathtt{t_j}))$,
$B_{26} = [G](\langle A\rangle \mathsf{tile} \rightarrow \bigvee_{up(t_i) = down(t_j)} (\langle A\rangle \mathtt{t_i} \wedge \langle A\rangle(\mathsf{bb} \wedge \langle A\rangle \mathtt{t_j})))$.

Lemma 6. *Given any finite set of tiles* $\mathcal{T}$*, the formula* $\Phi_{\mathcal{T}}$ *is satisfiable if and only if* $\mathcal{T}$ *can tile the second octant* $\mathcal{O}$.

As the model construction in the above proof can be carried out on any linear ordering containing an infinite ascending chain of points, Theorem 4 for the logic ADE immediately follows.

As for the other logics considered in the first half of Theorem 4, it suffices to modify the formulas involving $\langle D \rangle$ (see [7]) and the formulas B_{14}, B_{16}, and B_{18}, which involve $\langle E \rangle$. As an example, in the case of the logic $\mathsf{AD\overline{O}}$, formulas B_{14}, B_{16}, and B_{18} must be replaced with the following ones:

$$B'_{14} = [G](\langle A \rangle(\mathsf{tile} \wedge \langle A \rangle \mathsf{tile}) \leftrightarrow \langle A \rangle(\mathsf{tile} \wedge \langle \overline{O} \rangle \mathsf{bb})),$$
$$B'_{16} = [G](\langle A \rangle(\mathsf{tile} \wedge \langle A \rangle \mathsf{tile}) \leftrightarrow \langle A \rangle(\mathsf{tile} \wedge \langle A \rangle\langle \overline{O} \rangle \mathsf{be})),$$
$$B'_{18} = [G](\langle A \rangle(\mathsf{tile} \wedge \langle A \rangle \mathsf{tile}) \leftrightarrow \langle A \rangle(\mathsf{tile} \wedge \langle \overline{O} \rangle \mathsf{eb})).$$

In the cases of the fragments where A is replaced with $\overline{\mathsf{A}}$ and E (resp., $\overline{\mathsf{E}}$) is replaced with B (resp., $\overline{\mathsf{B}}$), the proof is perfectly symmetric and it takes advantage of the existence of an infinite descending sequence.

4.3 Reduction from the $\mathbb{N} \times \mathbb{N}$ Tiling Problem

In this section, we strengthen some of the results of Theorem 3 by showing that the satisfiability problem for the fragments $\mathsf{B\overline{E}}$, $\mathsf{B\overline{E}}$, and $\mathsf{\overline{B}\overline{E}}$ is undecidable (the case of BE was already dealt with by Theorem 2 for the classes Lin and Den). The proof is based on a reduction from the $\mathbb{N} \times \mathbb{N}$ tiling problem, which is a non-trivial adaptation of the reduction from the same problem provided by Marx and Reynolds to prove the undecidability of Compass Logic [19].

Theorem 5. *The satisfiability problem for* $\mathsf{B\overline{E}}$ *(respectively,* $\mathsf{B\overline{E}}$*) is undecidable in any class of linear orders that contains a linear order with an infinite ascending (respectively, descending) chain. The satisfiability problem for* $\mathsf{\overline{B}\overline{E}}$ *is undecidable in any class of linear orders that contains a linear order with an infinite chain indexed by the integers.*

The encoding of the quadrant $\mathbb{N} \times \mathbb{N}$ is close to that given in [19] (it is based on a suitable enumeration of its elements). From such a work, we also borrow the set of propositional variables p, q, right, left, above, floor, and wall used in the proof.

Hereafter, we restrict ourselves to the easiest case of $\mathsf{B\overline{E}}$ (however, the proof can be adapted to the other two fragments). The operators of $\mathsf{B\overline{E}}$ can be naturally mapped into those of Compass Logic as follows: if $\mathbf{M}, [a, b] \Vdash \langle \overline{B} \rangle \psi$, then $\mathbf{M}, [a, c] \Vdash \psi$ for some $c > b$ and thus $\langle \overline{B} \rangle$ corresponds to $\Diamond$ in Compass Logic, and if $\mathbf{M}, [a, b] \Vdash \langle E \rangle \psi$, then $\mathbf{M}, [c, b] \Vdash \psi$ for some $a < c \leq b$ and thus $\langle E \rangle$ corresponds to $\Diamond\!\!\!\!-$.

First, we define the *always in the future* operator $[G]$:

$$[G]\varphi = \varphi \wedge [E]\varphi \wedge [\overline{B}](\varphi \wedge [E]\varphi).$$

The properties of p and q, that respectively encode the elements of the quadrant and the successor relation over them (with respect to the given enumeration), are expressed by the following formulas:

$$
\begin{aligned}
&N_1 = \mathsf{p},\\
&N_2 = [G](\mathsf{p} \rightarrow [\overline{B}]\neg\mathsf{p}),\\
&N_3 = [G](\mathsf{p} \rightarrow [E]\neg\mathsf{p}),\\
&N_4 = [G](\langle\overline{B}\rangle\mathsf{p} \rightarrow [E]\neg\mathsf{p}),\\
&N_5 = [G](\mathsf{p} \rightarrow [E]([\overline{B}]\neg\mathsf{p}),\\
&N_6 = [G](\mathsf{q} \rightarrow [\overline{B}]\neg\mathsf{q}),\\
&N_7 = [G](\mathsf{p} \rightarrow \langle\overline{B}\rangle\mathsf{q}),\\
&N_8 = [G](\mathsf{q} \rightarrow \langle E\rangle\mathsf{p}),\\
&N_9 = [G](\langle\overline{B}\rangle\mathsf{q} \rightarrow [E]\neg\mathsf{p}).
\end{aligned}
$$

As an immediate consequence from N_1-N_9, we have:

$$N_{10} = [G](\mathsf{q} \rightarrow [\overline{B}]\neg\mathsf{p}).$$

The above formulas state that both p and q are injective functions, that is, if $\mathbf{M}, [a,b] \Vdash \mathsf{p}$, then for each $c \neq b$ $\mathbf{M}, [a,c] \Vdash \neg\mathsf{p}$ and for each $d \neq a$ $\mathbf{M}, [d,b] \Vdash \neg\mathsf{p}$, and similarly for q, that p-intervals cannot be subintervals of p-intervals (and they do not overlap), that q and p have the same domain and range, that is, $\mathbf{M}, [a,b] \Vdash \mathsf{p}$ if and only if there exists $c > b$ such that $\mathbf{M}, [a,c] \Vdash \mathsf{q}$ and $\mathbf{M}, [a,b] \Vdash \mathsf{p}$ if and only if there exists $c < a$ such that $\mathbf{M}, [c,b] \Vdash \mathsf{q}$, and, finally, that a p-interval cannot be a subinterval of a q-interval.

Lemma 7. *For every model* $\mathbf{M}$ *and every interval* $[a,b]$ *such that* $\mathbf{M}, [a,b] \Vdash N_1 \wedge \ldots \wedge N_9$ *there exists a sequence of intervals* $[a,b] = [a_0,b_0], [a_1,b_1], \ldots$ *such that, for every* $n \geq 0$*: (1)* $b_n \leq a_{n+1}$*; (2)* $\mathbf{M}, [a_n,b_n] \Vdash \mathsf{p}$*; (3)* $\mathbf{M}, [a_n,b_{n+1}] \Vdash \mathsf{q}$*; (4) if* $\mathbf{M}, [a',b'] \Vdash \mathsf{p}$ *and* $b_0 \leq b' < b_n$*, then there exists* $m < n$ *such that* $[a',b'] = [a_m,b_m]$.

Lemma 7 corresponds to Claim 5.2, Section 5.4 in [19]. To prove Claim 5.3, we translate formulas A_6-A_{18} in [19] to the language $\overline{\mathrm{B}}\mathrm{E}$. For a given formula φ, let $F(\varphi)$ be the conjunction of the following formulas:

$$
\begin{aligned}
&[G](\varphi \rightarrow [\overline{B}]\neg\varphi),\\
&[G](\varphi \rightarrow [E]\neg\varphi),\\
&[G](\mathsf{p} \rightarrow \langle\overline{B}\rangle\varphi),\\
&[G](\varphi \rightarrow \langle E\rangle\mathsf{p}),\\
&[G](\mathsf{q} \rightarrow [E](\langle\overline{B}\rangle\varphi \rightarrow \mathsf{p}).
\end{aligned}
$$

The above formulas state that φ is an injective function, that the domain of p is included in the domain of φ, that the range of φ is included in the range of p, and that the domain of φ is included in the domain of p (that is, the domain of p and that of φ coincide).

Formulas A_6-A_{18} can be encoded as follows:

$$
\begin{aligned}
A_6 &= F(\mathsf{right}),\\
A_7 &= F(\mathsf{above}),\\
A_8 &= [G](\langle \overline{B}\rangle \mathsf{right} \rightarrow [E]\neg \mathsf{right}),\\
A_9 &= [G](\mathsf{right} \rightarrow \langle \overline{B}\rangle \mathsf{above}),\\
A_{10} &= [G](\mathsf{right} \rightarrow [\overline{B}](\langle \overline{B}\rangle \mathsf{above} \rightarrow [E]\neg \mathsf{p})),
\end{aligned}
$$

which impose that both right and above are total injective functions from p-intervals to p-intervals, that right is strictly monotone, and that above is the composition of right and q, and:

$$
\begin{aligned}
A_{11} &= \mathsf{floor} \wedge \mathsf{wall},\\
A_{12} &= [\overline{B}]\neg(\mathsf{floor} \wedge \mathsf{wall}) \wedge [E]\neg(\mathsf{floor} \wedge \mathsf{wall}) \wedge [\overline{B}][E]\neg(\mathsf{floor} \wedge \mathsf{wall}),\\
A_{13} &= [G]((\mathsf{floor} \vee \mathsf{wall}) \rightarrow \mathsf{p}),\\
A_{14} &= [G](\mathsf{wall} \rightarrow [\overline{B}](\mathsf{q} \rightarrow [E](\mathsf{p} \rightarrow \mathsf{floor}))),\\
A_{15} &= [G](\mathsf{wall} \rightarrow \langle \overline{B}\rangle(\mathsf{above} \wedge \langle E\rangle \mathsf{wall})),\\
A_{16} &= [G]((\mathsf{p} \wedge \neg \mathsf{wall}) \rightarrow [\overline{B}](\mathsf{above} \rightarrow [E]\neg \mathsf{wall})),\\
A_{17} &= [G](\mathsf{right} \rightarrow [E]\neg \mathsf{wall}),\\
A_{18} &= [\overline{B}](\langle E\rangle(\mathsf{p} \wedge \neg \mathsf{wall}) \rightarrow \mathsf{right} \vee \langle E\rangle(\mathsf{right} \wedge \langle E\rangle(\mathsf{p} \wedge \neg \mathsf{wall}))),
\end{aligned}
$$

which state the properties of floor and wall. Intuitively, we have the following properties: the initial interval is labeled with floor and wall and this is not the case with any other interval; both floor and wall are p-intervals; the successor of a wall is a floor; above every wall there is a wall, and, with the exception of the initial interval, every wall is above a wall; right never goes to the wall, and every non-wall p-interval has a p-interval on the left.

Finally, let $\phi_{\mathcal{T}}$ be the conjunction of formulas N_1-N_9, A_6-A_{18}, and A_{19}-A_{22} below:

$$
\begin{aligned}
A_{19} &= [G](\mathsf{p} \leftrightarrow \textstyle\bigvee_{i=1}^{k} \mathtt{t_i}),\\
A_{20} &= [G] \textstyle\bigwedge_{i \neq j} \neg(\mathtt{t_i} \wedge \mathtt{t_j}),\\
A_{21} &= \textstyle\bigwedge_{up(t_i) \neq down(t_j)} [G]\neg(\mathtt{t_i} \wedge \langle \overline{B}\rangle(\mathsf{above} \wedge \langle E\rangle \mathtt{t_j})),\\
A_{22} &= \textstyle\bigwedge_{right(t_i) \neq left(t_j)} [G]\neg(\mathtt{t_i} \wedge \langle \overline{B}\rangle(\mathsf{right} \wedge \langle E\rangle \mathtt{t_j})).
\end{aligned}
$$

The proof of the next lemma repeats, *mutatis mutandis*, the one in [19].

Lemma 8. *A set of tiles $\mathcal{T}$ can tile $\mathbb{N} \times \mathbb{N}$ if and only if $\phi_{\mathcal{T}}$ is satisfiable.*

This concludes the proof of Theorem 5 for the case $\overline{\mathsf{B}}\mathsf{E}$. A similar construction can be carried out for the logics $\overline{\mathsf{B}\mathsf{E}}$ and $\mathsf{B}\overline{\mathsf{E}}$. As for $\overline{\mathsf{B}\mathsf{E}}$, it suffices to replace the first quadrant with the second one, where the operator $\langle \overline{B}\rangle$ corresponds to the operator $\diamondsuit$ and the operator $\langle \overline{E}\rangle$ corresponds to the operator $\Diamond$ of Compass Logic. As for $\mathsf{B}\overline{\mathsf{E}}$, the construction of the model is obtained in the third quadrant instead of the second one.

5 Concluding Remarks

In this paper, we have taken into consideration the variety of HS fragments that can be obtained by choosing suitable subsets of the set of the twelve basic modal operators (corresponding to Allen's relations) extended with two additional operators for subintervals. We have focused our attention on the problem of classifying them with respect to decidability/undecidability (first raised by Halpern and Shoham in [15], Problem 3). Besides a summary of the state of the art, we have given a number of new undecidability results based on suitable reductions from tiling problems.

The proposed classification is naturally related to definability/undefinability relations among operators. Known definability relations reduce the number of fragments from over 16 thousands to less than 5 thousands, and the results reported in this paper cover more than half of these cases. Our study not only makes a substantial contribution to the complete solution of the classification problem inherited from [15], but it also suggests some directions to explore in the search of other decidable interval logics.

It is worth pointing out that all undecidability results reported here hinge on the existence of an infinite ascending/descending chain of intervals. Decidability problems for interval logics over finite interval structures are still largely unexplored. Some positive results for PNL can be found in [6,7,8,9,10].

References

1. Allen, J.F.: Maintaining knowledge about temporal intervals. Communications of the ACM 26(11), 832–843 (1983)
2. Börger, E., Grädel, E., Gurevich, Y.: The Classical Decision Problem. In: Perspectives of Mathematical Logic. Springer, Heidelberg (1997)
3. Bresolin, D., Goranko, V., Montanari, A., Sala, P.: Tableau-based Decision Procedure for the Logic of Proper Subinterval Structures over Dense Orderings. In: Areces, C., Demri, S. (eds.) Proceedings of the 5th International Workshop on Methods for Modalities (M4M), pp. 335–351 (2007)
4. Bresolin, D., Goranko, V., Montanari, A., Sala, P.: Tableau systems for logics of subinterval structures over dense orderings. In: Olivetti, N. (ed.) TABLEAUX 2007. LNCS (LNAI), vol. 4548, pp. 73–89. Springer, Heidelberg (2007)
5. Bresolin, D., Goranko, V., Montanari, A., Sala, P.: Tableau-based decision procedures for the logics of subinterval structures over dense orderings. In: Journal of Logic and Computation (to appear, 2008)
6. Bresolin, D., Goranko, V., Montanari, A., Sciavicco, G.: On decidability and expressiveness of propositional interval neighborhood logics. In: Artemov, S.N., Nerode, A. (eds.) LFCS 2007. LNCS, vol. 4514, pp. 84–99. Springer, Heidelberg (2007)
7. Bresolin, D., Goranko, V., Montanari, A., Sciavicco, G.: Propositional Interval Neighborhood Logics: Expressiveness, Decidability, and Undecidable Extensions. Annals of Pure and Applied Logic (to appear, 2008)
8. Bresolin, D., Montanari, A., Sala, P.: An optimal tableau-based decision algorithm for propositional neighborhood logic. In: Thomas, W., Weil, P. (eds.) STACS 2007. LNCS, vol. 4393, pp. 549–560. Springer, Heidelberg (2007)

9. Bresolin, D., Montanari, A., Sala, P., Sciavicco, G.: Optimal Tableaux for Right Propositional Neighborhood Logic over Linear Orders. In: JELIA 2008. LNCS (LNAI), vol. 5293, pp. 62–75. Springer, Heidelberg (2008)
10. Bresolin, D., Montanari, A., Sciavicco, G.: An optimal decision procedure for Right Propositional Neighborhood Logic. Journal of Automated Reasoning 38(1-3), 173–199 (2007)
11. Chaochen, Z., Hoare, C.A.R., Ravn, A.P.: A calculus of durations. Information Processing Letters 40(5), 269–276 (1991)
12. Emerson, E.A.: Temporal and modal logic. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, Formal Models and Semantics, vol. B, pp. 995–1072. MIT Press, Cambridge (1990)
13. Goranko, V., Montanari, A., Sciavicco, G.: Propositional interval neighborhood temporal logics. Journal of Universal Computer Science 9(9), 1137–1167 (2003)
14. Goranko, V., Montanari, A., Sciavicco, G.: A road map of interval temporal logics and duration calculi. Journal of Applied Non-Classical Logics 14(1–2), 9–54 (2004)
15. Halpern, J., Shoham, Y.: A propositional modal logic of time intervals. Journal of the ACM 38(4), 935–962 (1991)
16. Hodkinson, I., Montanari, A., Sciavicco, G.: Non-finite axiomatizability and undecidability of interval temporal logics with C, D, and T. In: Kaminski, M., Martini, S. (eds.) CSL 2008. LNCS, vol. 5213, pp. 308–322. Springer, Heidelberg (2008)
17. Krokhin, A.A., Jeavons, P., Jonsson, P.: Reasoning about temporal relations: The tractable subalgebras of Allen's interval algebra. Journal of the ACM 50(5), 591–640 (2003)
18. Lodaya, K.: Sharpening the undecidability of interval temporal logic. In: He, J., Sato, M. (eds.) ASIAN 2000. LNCS, vol. 1961, pp. 290–298. Springer, Heidelberg (2000)
19. Marx, M., Reynolds, M.: Undecidability of compass logic. Journal of Logic and Computation 9(6), 897–914 (1999)
20. Montanari, A., Sciavicco, G., Vitacolonna, N.: Decidability of interval temporal logics over split-frames via granularity. In: Flesca, S., Greco, S., Leone, N., Ianni, G. (eds.) JELIA 2002. LNCS, vol. 2424, pp. 259–270. Springer, Heidelberg (2002)
21. Moszkowski, B.: Reasoning about digital circuits. Tech. rep. stan-cs-83-970, Dept. of Computer Science, Stanford University, Stanford, CA (1983)
22. Otto, M.: Two variable first-order logic over ordered domains. Journal of Symbolic Logic 66(2), 685–702 (2001)
23. Venema, Y.: Expressiveness and completeness of an interval tense logic. Notre Dame Journal of Formal Logic 31(4), 529–547 (1990)

The Variable Hierarchy for the Lattice μ-Calculus*

Walid Belkhir and Luigi Santocanale

Laboratoire d'Informatique Fondamentale de Marseille
Université de Provence

Abstract. The *variable hierarchy problem* asks whether every μ-term t is equivalent to a μ-term t' where the number of fixed-point variables in t' is bounded by a constant. In this paper we prove that the variable hierarchy of the *lattice μ-calculus* – whose standard interpretation is over the class of all complete lattices – is infinite, meaning that such a constant does not exist if the μ-terms are built up using the basic lattice operations as well as the least and the greatest fixed point operators. The proof relies on the description of the lattice μ-calculus by means of games and strategies.

1 Introduction

Hierarchies and expressivity issues are at the core of fixed-point theory [1,2]. The alternation depth and the star height hierarchies have been intensively studied [3,4,5,6,7,8] In this paper we deal with a refinement of the star height hierarchy, the variable hierarchy introduced in [9], and consider it within the *lattice μ-calculus* $\mathbb{L}_\mu$. The latter is a concrete representation of the theory of binary infs and sups, and of least and greatest fixed points over complete lattices.

Let us recall the background of the lattice μ-calculus. Two-players games are a standard model for the possible interactions between a system and its potentially adverse environment [10,11,12]. It was proposed in [13] to develop a theory of communication grounded on similar game theoretic ideas and, moreover, on algebraic concepts such as "free lattice" [14] and "free bicomplete category" [15]. A first work pursued this idea using tools of categorical logic [16]. The proposal was further developed in [17] where cycles were added to lattice terms to enrich the model with possibly infinite behaviors. As a result, lattice terms were replaced by lattice μ-terms and their combinatorial representation, *parity games*. Given two parity games G, H the witness that the relation $G \leq H$ holds in every complete lattice interpretation is a winning strategy for a prescribed player, Mediator, in a game $\langle G, H\rangle$. A game G may also be considered as modelling a synchronous communication channel available to two users. Then, a winning strategy for Mediator in $\langle G, H\rangle$ witnesses the existence of an asynchronous protocol allowing one user of G to communicate with the other user on H ensuring absence of deadlocks.

* Research supported by the *Agence Nationale de la Recherche*, project SOAPDC.

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 605–620, 2008.

Apart from its primary goal, that of describing complete lattices, a major interest of this μ-calculus stems from its neat proof-theory, a peculiarity within the theory of fixed-point logics. The idea that winning strategies for Mediator in the game $\langle G, H\rangle$ are sort of circular proofs was formalized in [18]. More interestingly, proof theoretic ideas and tools – the cut elimination procedure and η-expansion, in their game theoretic disguise – have proved quite powerful to solve deep problems arising from fixed-point theory. These are the alternation-depth hierarchy problem [5] and the status of the ambiguous classes [19]. The same ideas shall be of help in establishing the strictness of the variable hierarchy.

The *variable hierarchy problem* of $\mathbb{L}_\mu$ asks whether every μ-term t is equivalent to a μ-term s where the number of fixed-point variables in s is bounded by a constant. This amounts to consider the classes

$$\mathcal{T}_n = \{\, t \in \mathbb{L}_\mu \mid t \sim s \text{ for some } s \in \mathbb{L}_\mu \text{ s.t. } nbr(s) \leq n \,\} \tag{1}$$

– where $\sim$ denotes semantic equivalence of μ-terms over complete lattices and $nbr(s)$ is the number of bound variables in s – and to ask whether the hierarchy made up of the $\mathcal{T}_n$ collapses: *is there a constant k such that $\mathcal{T}_n = \mathcal{T}_k$ for every $n \geq k$?* We shall work through several combinatorial refinements of the original problem: on the one hand we consider the representation of μ-terms by means of parity games, on the other hand we need two digraph complexity measures, the *feedback* and the *entanglement*, that roughly speaking compute the minimal number of bound variables of a μ-term up to α-conversion. The question about the classes $\mathcal{T}_n$ shall be answered in the negative when showing that the analogous question for the classes

$$\mathcal{L}_n = \{\, G \in \mathcal{G} \mid G \sim H \text{ for some } H \in \mathcal{G} \text{ s.t. } \mathcal{E}(H) \leq n \,\}$$

has a negative answer. Here $\mathcal{G}$ is the collection of parity games with draw positions, $\sim$ denotes semantic equivalence, and $\mathcal{E}(H)$ is the entanglement of the graph of positions and moves of H. We shall construct, for each $n \geq 1$, a parity game G_n with two properties: (i) G_n has entanglement n, showing that G_n belongs to $\mathcal{L}_n$, (ii) G_n is semantically equivalent to no game in $\mathcal{L}_{n-3}$. Thus, we shall prove that the inclusions $\mathcal{L}_{n-3} \subseteq \mathcal{L}_n$, $n \geq 3$, are strict.

The key ideas in our proof are as follows. The games G_n are *strongly synchronizing*.[1] By playing with the η-expansion – i.e. the copycat strategy – and the cut-elimination – i.e. composition of strategies – we prove that the syntactical structure of a game H, semantically equivalent to a strongly synchronizing game G, resembles that of G: every move (edge) in G can be simulated by a non empty finite sequence of moves (a path) of H; if two paths simulating distinct edges do intersect, then the edges do intersect as well. We formalize such situation within the notion of $\star$-*weak simulation*. The relevant result is that if there is a $\star$-weak simulation of G by H, then $\mathcal{E}(G) - 2 \leq \mathcal{E}(H)$.

[1] A synchronizing game [5] has the property that there exists just one winning strategy for Mediator in $\langle G, G\rangle$, the copycat strategy. We need here to strengthen this notion.

The paper is organized as follows. Section 2 introduces the lattice μ-calculus: its syntax, its semantics, the translation of μ-terms into parity games, the canonical preorder. In Section 3, we firstly recall the definition of entanglement and reduce the hierarchy problem for the classes $\mathcal{T}_n$ to that for the classes $\mathcal{L}_n$; then we define the $\star$-weak simulations between graphs. In Section 4, we define strongly synchronizing games and prove their hardness w.r.t. the variable hierarchy. In Section 5, we construct strongly synchronizing games of arbitrary entanglement. We sum up the discussion in our main result, Theorem 20.

Notation, preliminary definitions, elementary facts. If G is a graph, then a path in G is a sequence of the form $\pi = g_0 g_1 \dots g_n$ such that $(g_i, g_{i+1}) \in E_G$ for $0 \leq i < n$. A path is *simple* if $g_i \neq g_j$ for $i, j \in \{0, \dots, n\}$ and $i \neq j$. The integer n is the length of π, g_0 is the source of π, noted $\delta_0\pi = g_0$, and g_n is the target of π, noted $\delta_1\pi = g_n$. We denote by $\Pi^+(G)$ the set of simple non empty (i.e. of length greater than 0) paths in G. A pointed digraph $\langle V, E, v_0\rangle$ of root v_0, is a *tree* if for each $v \in V$ there exists a unique path from v_0 to v. A *tree with back-edges* is a tuple $\mathcal{T} = \langle V, T, v_0, B\rangle$ such that $\langle V, T, v_0\rangle$ is a tree, and $B \subseteq V \times V$ is a second set of edges such that if $(x, y) \in B$ then y is an ancestor of x in the tree $\langle V, T, v_0\rangle$. We shall refer to edges in T as tree edges and to edges in B as back-edges. We say that $r \in V$ is a return of $\mathcal{T}$ if there exists $x \in V$ such that $(x, r) \in B$. A return r is *active* in v if $(x, r) \in B$ for some for some descendant x of v. The *feedback of a vertex* v is the number of active returns of v. The *feedback of a tree with back-edges* is the maximum feedback of its vertices. We shall say that a pointed directed graph (V, E, v_0) is a tree with back-edges if there is a partition of E into two disjoint subsets T, B such that $\langle V, T, v_0, B\rangle$ is a tree with back-edges. If $\mathcal{T}$ is a tree with back-edges, then a path in $\mathcal{T}$ can be factored as $\pi = \pi_1 * \dots * \pi_n * \tau$, where each factor π_i is a sequence of tree edges followed by a back-edge, and τ does not contain back-edges. Such factorization is uniquely determined by the occurrences of back edges in π. For $i > 0$, let r_i be the return at the end of the factor π_i. Let also r_0 be the source of π. Let the b-length of π be the number of back-edges in π. i.e. $r_i = \delta_1\pi_i$.

Lemma 1. *If π is a simple path of b-length n, then r_n is the vertex closest to the root visited by π. Hence, if a simple path π lies in the subtree of its source, then it is a tree path.*

A *cover* or *unravelling* of a (finite) directed graph H is a (finite) graph K together with a surjective graph morphism $\psi : K \longrightarrow H$ such that for each $v \in V_K$, the correspondence sending k to $\rho(k)$ restricts to a bijection from $\{ k \in V_K \mid (v, k) \in E_K \}$ to $\{ h \in V_H \mid (\psi(v), h) \in E_H \}$.

2 The Lattice μ-Calculus $\mathbb{L}_\mu$

We introduce the lattice μ-calculus, denoted $\mathbb{L}_\mu$, and its standard semantics over complete lattices. One may consider its semantics over μ-lattices [17] as well.

Syntax of $\mathbb{L}_\mu$. The syntax of lattice μ-terms is given by the following grammar:

$$t \;=\; x \mid \top \mid \bot \mid t \wedge t \mid t \vee t \mid \mu x.t \mid \nu x.t\,,$$

where x ranges over X, a countable set of variables.

Semantics of $\mathbb{L}_\mu$ over complete lattices. If t is a μ-term, then we denote by X_t the set of free variables of t. Given a complete lattice L, we define the interpretation of a μ-term t as the function $\|t\|^L : L^{X_t} \longrightarrow L$, where L^X is the X-fold product lattice of L with itself, in the following way:

- If $t = x$, then $\|t\|^L(v) = v(x)$.
- If $t = \top$ (resp. $t = \bot$), then $\|t\|^L$ is interpreted as the constant function with value $\bigvee_L L$, i.e. the supremum of L (resp. to $\bigwedge_L L$, i.e. the infimum of L).
- If $t = t_1 \wedge t_2$, then $\|t\|^L(v) = \|t_1\|^L(v|t_1) \wedge_L \|t_2\|^L(v|t_2)$, where $\wedge_L$ denotes the *greatest lower bound* in L and $v|t_i$ is the restriction of v to X_{t_i}.
- If $t = t_1 \vee t_2$, then $\|t\|^L(v) = \|t_1\|^L(v|t_1) \vee_L \|t_2\|^L(v|t_2)$, where $\vee_L$ denotes the *least upper bound* in L.
- If $t = \mu x.t_1$, then we consider the monotone function $\phi(z) = \|t_1\|^L(v^z)$ – where $v^z(y) = v(y)$ if $y \neq x$ and $v^z(x) = z$ – and let $\|\mu x.t_1\|^L(v)$ be $\mu z.\phi(z)$, the least fixpoint of ϕ which exists and is unique in L by [20].
- Similar definition is given if $t = \nu x.t_1$, by substituting each symbol μ with the symbol ν, and the phrase *least fixpoint* with the phrase *greatest fixpoint*.

The variable hierarchy problem of $\mathbb{L}_\mu$. Let $s, t \in \mathbb{L}_\mu$, we write $nbr(s)$ for the number of bound variables in s and $t \sim s$ to mean that $\|t\|^L = \|s\|^L$ holds in every complete lattice L. Recall from Equation (1) the definition of the classes $\mathcal{T}_n$. We ask whether the hierarchy made up of these classes collapses: is there a constant k such that $\mathcal{T}_n = \mathcal{T}_k$ for every $n \geq k$?

Lattice μ-terms have a natural translation into a kind of 2-players games: the *labeled parity games with draws*. We first define these games, then we provide such translation.

Labeled parity games with draws. A *labeled parity game with draws* is a tuple $G = \langle Pos_E^G, Pos_A^G, Pos_D^G, M^G, \rho^G, p_\star^G, \lambda^G \rangle$ where:

- $Pos_E^G, Pos_A^G, Pos_D^G$ are finite pairwise disjoint sets of positions (Eva's positions, Adam's positions, and draw positions).
- M^G, the set of moves, is a subset of $(Pos_E^G \cup Pos_A^G) \times (Pos_E^G \cup Pos_A^G \cup Pos_D^G)$,
- ρ^G is a mapping from $(Pos_E^G \cup Pos_A^G)$ to $\mathbb{N}$.
- $p_\star^G \in Pos_E^G \cup Pos_A^G \cup Pos_D^G$ is the initial position.
- $\lambda^G : Pos_D^G \to X$ is a labelling of draw positions with variables.

These data define a game between player Eva and player Adam starting from the initial position. The outcome of a finite play is determined according to the normal play condition: a player who cannot move loses. It can also be a draw, if a position in Pos_D^G is reached. The outcome of an infinite play $\{ (g_k, g_{k+1}) \in M^G \}_{k \geq 0}$ is determined by means of the rank function ρ^G as follows: it is a win for Eva iff the maximum of the set $\{ i \in \mathbb{N} \mid \exists$ infinitely many k s.t. $\rho^G(g_k) = i \}$

is even. To simplify the notation, we shall use $Pos^G_{E,A}$ for the set $Pos^G_E \cup Pos^G_A$ and use similar notations such as $Pos^G_{E,D}$, etc. We let $Max^G = \max \rho^G(Pos^G_{E,A})$ if the set $Pos^G_{E,A}$ is not empty, and $Max^G = -1$ otherwise. We denote by (G, g) the game that differs from G only on the starting position, i.e. $p_\star^{(G,g)} = g$, and similarly we write (G, g) to mean that the play has reached position g. With $\mathcal{G}$ we shall denote the collection of all labeled parity games; as no confusion will arise, we will call a labeled parity game with simply "game".

Translation of μ-terms into games. The translation $\gamma : \mathbb{L}_\mu \to \mathcal{G}$ is so defined:

- If $t = x$, then $\gamma(t) = \hat{x}$, where $\hat{x}$ is the game with just one final draw position, of zero priority, labeled with variable x.
- If $t = \top$, (resp. $t = \bot$) then $\gamma(t)$ is the game with just one position, of zero priority, which belongs to Adam (resp. to Eva).
- If $t = t_1 \wedge t_2$, then the game $\gamma(t)$ is obtained from the games $G_i = \gamma(t_i)$, $i = 1, 2$, by adding to the disjoint union $Pos^{G_1}_{E,A,D} \uplus Pos^{G_2}_{E,A,D}$ a new initial position $p_\star^{\gamma(t)} \in Pos_A^{\gamma(t)}$ such that $\rho^{\gamma(t)}(p_\star^{\gamma(t)}) = 0$; moreover the moves $(p_\star^{\gamma(t)}, p_\star^{G_i})$, $i = 1, 2$, are added to $M^{G_1} \cup M^{G_2}$.
- If $t = t_1 \vee t_2$, then we define $\gamma(t)$ as above, apart that $p_\star^{\gamma(t)} \in Pos_E^{\gamma(t)}$.
- For $t = \theta x.t_1, \theta \in \{ \mu, \nu \}$, assume that $G = \gamma(t_1)$. Let $p_\star^{\gamma(t)}$ be a new initial position. Let also $Pos_x = \{ g \in Pos^G_D \mid \lambda^G(g) = x \}$ and $Pred_x = \{ g \in Pos^G_{E,A} \mid (g, g') \in M^G$ and $g' \in Pos_x \}$. Then we define $\gamma(t)$ as follows:
  - $Pos_E^{\gamma(t)} = Pos^G_E \cup \{ p_\star^{\gamma(t)} \}$ if $\theta = \mu$ and $Pos_E^{\gamma(t)} = Pos^G_E$ if $\theta = \nu$.
  - $Pos_A^{\gamma(t)} = Pos^G_A$ if $\theta = \mu$ and $Pos_A^{\gamma(t)} = Pos^G_A \cup \{ p_\star^{\gamma(t)} \}$ if $\theta = \nu$.
  - $Pos_D^{\gamma(t)} = Pos^G_D \setminus Pos_x$.
  - $M^{\gamma(t)} = M^G_{|Pos_{E,A,D}^{\gamma(t)}} \cup \{ (g, p_\star^{\gamma(t)}) \mid g \in Pred_x \} \cup \{ (p_\star^{\gamma(t)}, p^G_\star) \}$.
  - $\rho^{\gamma(t)}$ is the extension of ρ^G to $p_\star^{\gamma(t)}$ as follows:
    - If $\theta = \mu$, then $\rho^{\gamma(t)}(p_\star^{\gamma(t)}) = Max^G$ if Max^G is odd, and $\rho^{\gamma(t)}(p_\star^{\gamma(t)}) = Max^G + 1$ if Max^G is even.
    - If $\theta = \nu$, then $\rho^{\gamma(t)}(p_\star^{\gamma(t)}) = Max^G$ if Max^G is even, and $\rho^{\gamma(t)}(p_\star^{\gamma(t)}) = Max^G + 1$ if Max^G is odd.

Let us emphasize that not every game G is of the form $\gamma(t)$ for some $t \in \mathbb{L}_\mu$ – for example, the graph of positions and moves of $\gamma(t)$ is a tree with back-edges whose returns have a unique successor. It is often convenient to consider vectorial versions of μ-calculi [2, §1.4.4, §2.7]. A vectorial μ-term can be understood as a *system of equations* of form $\{ x_i =_{\theta_i} f_i(x_{i,1}, \ldots, x_{i,n_i}) \}_{i \in 1,\ldots,k}$ with $\theta_i \in \{ \mu, \nu \}$. The step that constructs a canonical solution of a system of equations by means of scalar μ-terms is known as the Bekič principle [2, §1.4.2]. The principle implies that the two alternatives of a μ-calculus, scalar or vectorial, are expressively equivalent. Games in $\mathcal{G}$ correspond to terms of a vectorial version of the μ-calculus $\mathbb{L}_\mu$: it should not be difficult for the reader to guess a translation from systems of equations in the signature $\top, \wedge, \bot, \vee$ to games.

The preorder on $\mathcal{G}$. In order to describe a preorder on the class $\mathcal{G}$, we define next a new game $\langle G, H\rangle$ for a pair of games G and H in $\mathcal{G}$. This is not a parity game with draws; to emphasize this fact, the two players are named Mediator and Opponents instead of Eva and Adam.

Definition 2. *The game $\langle G, H\rangle$ is defined as follows:*

- *The set of Mediator's positions is* $Pos^G_A \times Pos^H_{E,D} \cup Pos^G_{A,D} \times Pos^H_E \cup \{\,(g,h) \in Pos^G_D \times Pos^H_D \mid \lambda^G(g) \neq \lambda^H(h)\,\}$ *and the set of Opponents' positions is* $Pos^G_E \times Pos^H_{E,A,D} \cup Pos^G_{E,A,D} \times Pos^H_A \cup \{\,(g,h) \in Pos^G_D \times Pos^H_D \mid \lambda^G(g) = \lambda^H(h)\,\}$
- *Moves of $\langle G, H\rangle$ are either left moves $(g,h) \to (g',h)$, where $(g,g') \in M^G$, or right moves $(g,h) \to (g,h')$, where $(h,h') \in M^H$; however the Opponents can play only with* Eva *on G or with Adam on H.*
- *A finite play is a loss for the player who can not move. An infinite play γ is a win for Mediator if and only if its left projection $\pi_G(\gamma)$ is a win for Adam, or its right projection $\pi_H(\gamma)$ is a win for Eva.*

Definition 3. *If G and H belong to $\mathcal{G}$, then we declare that $G \leq H$ if Mediator has a winning strategy in the game $\langle G, H\rangle$ starting from position $(p^G_\star, p^H_\star)$.*

The following is the reason to consider such a syntactic relation:

Theorem 4 (See [17]). *The relation $\leq$ is sound and complete with respect to the interpretation in any complete lattice, i.e. $\gamma(t_1) \leq \gamma(t_2)$ if and only if $\|t_1\|^L \leq \|t_2\|^L$ holds in every complete lattice L.*

In the sequel we shall write $G \sim H$ to mean that $G \leq H$ and $H \leq G$; notice that this coherently subsumes our previous usage of $\sim$. It was proved in [17] that $G \leq G$, by exhibiting the *copycat* strategy in the game $\langle G, G\rangle$: from a position (g,g), it is Opponents' turn to move either on the left or on the right board. When they stop moving, Mediator will have the ability to copy all the moves played by the Opponents so far to the opposite board until the play reaches the position (g',g'). There it was also proved that if $G \leq H$ and $H \leq K$ then $G \leq K$, a sort of cut-elimination Theorem. This result was achieved by describing a game $\langle G, H, K\rangle$ with the following properties: *(i)* given two winning strategies R on $\langle G, H\rangle$, and S on $\langle H, K\rangle$ there is a winning strategy $R\|S$ on $\langle G, H, K\rangle$, that is the composition of the strategies R and S, *(ii)* given a winning strategy T on $\langle G, H, K\rangle$, there exists a winning strategy $T_{\setminus H}$ on $\langle G, K\rangle$.

Definition 5. *Positions of the game $\langle G, H, K\rangle$ are triples $(g,h,k) \in Pos^G_{A,E,D} \times Pos^H_{A,E,D} \times Pos^K_{A,E,D}$ such that*

- *the set of Mediator's positions is* $Pos^G_A \times Pos^H_{A,E,D} \times Pos^K_{E,D} \cup Pos^G_{A,D} \times Pos^H_{A,E,D} \times Pos^K_E \cup \mathcal{L}(M)$, *and the set of Opponents' positions is* $Pos^G_E \times Pos^H_{A,E,D} \times Pos^K_{E,A,D} \cup Pos^G_{E,A,D} \times Pos^H_{A,E,D} \times Pos^K_A \cup \mathcal{L}(O)$, *where $\mathcal{L}(M), \mathcal{L}(O) \subseteq Pos^G_D \times Pos^H_{A,E,D} \times Pos^K_D$ are positions of Mediator and*

Opponents, respectively, defined as follows. Whenever $(g,h,k) \in Pos^G_D \times Pos^H_{A,E,D} \times Pos^K_D$, *then if* $h \in Pos^H_{E,A}$, *then the position* (g,h,k) *belongs to Mediator, otherwise, i.e.* $h \in Pos^H_D$, *then the final position* (g,h,k) *belongs to Opponents if and only if* $\lambda^G(g) = \lambda^H(h) = \lambda^K(k)$.

- *Moves of* $\langle G,H,K\rangle$ *are either left moves* $(g,h,k) \to (g',h,k)$ *where* $(g,g') \in M^G$ *or central moves* $(g,h,k) \to (g,h',k)$, *where* $(h,h') \in M^H$, *or right moves* $(g,h,k) \to (g,h,k')$, *where* $(k,k') \in M^K$; *however the Opponents can play only with Eva on* G *or with Adam on* K.
- *As usual, a finite play is a loss for the player who cannot move. An infinite play* γ *is a win for Mediators if and only if* $\pi_G(\gamma)$ *is a win for Adam on* G, *or* $\pi_K(\gamma)$ *is a win for Eva on* K.

3 Entanglement and $\star$-Weak Simulations

In order to compute the minimum number of bound variables required in a vectorial μ-term, a digraph measure called *entanglement* is needed. Its definition is as follows: the *entanglement* of a digraph G is *the minimum feedback of the finite unravellings of* G *into a tree with back-edges*. In [21], the entanglement of G has been characterized by means of a game $\mathcal{E}(G,k)$, $k = 0,\ldots,|V_G|$, played by Thief against Cops, a team of k cops.

Definition 6. *The entanglement game* $\mathcal{E}(G,k)$ *of a digraph* G *is defined by:*

- *Its positions are of the form* (v,C,P), *where* $v \in V_G$, $C \subseteq V_G$ *and* $|C| \leq k$, $P \in \{Cops, Thief\}$.
- *Initially Thief chooses* $v_0 \in V_G$ *and moves to* $(v_0,\emptyset,Cops)$.
- *Cops can move from* $(v,C,Cops)$ *to* $(v,C',Thief)$ *where* C' *can be*
  - C *: Cops skip,*
  - $C \cup \{\, v \,\}$ *: Cops add a new Cop on the current position,*
  - $(C \setminus \{\, x \,\}) \cup \{\, v \,\}$ *: Cops move a placed Cop to the current position.*
- *Thief can move from* $(v,C,Thief)$ *to* $(v',C,Cops)$ *if* $(v,v') \in E_G$ *and* $v' \notin C$.

Every finite play is a win for Cops, and every infinite play is a win for Thief.

The following will constitute our working definition of entanglement: $\mathcal{E}(G)$, *the entanglement of* G, *is the minimum* $k \in \{\, 0,\ldots,|V_G|\,\}$ *such that Cops have a winning strategy in* $\mathcal{E}(G,k)$.
The following proposition provides a useful variant of entanglement games.

Proposition 7. *Let* $\widetilde{\mathcal{E}}(G,k)$ *be the game played as the game* $\mathcal{E}(G,k)$ *apart that Cops is allowed to retire a number of cops placed on the graph. Then Cops has a winning strategy in* $\mathcal{E}(G,k)$ *if and only if he has a winning strategy in* $\widetilde{\mathcal{E}}(G,k)$.

A combinatorial refinement of the variable hierarchy problem. Along this paper, when referring to the entanglement or to the feedback of a game $G \in \mathcal{G}$, we mean the entanglement of the *its underlying graph*, i.e. the of graph of positions and moves. If we need to emphasize the distinction between a game G and its underlying graph, then we shall use $\overline{G}$ to denote the latter. The aim

is now to argue that the variable hierarchy problem of $\mathbb{L}_\mu$ resolves to a problem on the entanglement of parity games with draws. Recall that $nbr(t)$ denotes the number of bound variables in t, and let us use $fb(T)$ to denote the feedback of a tree with back-edges T. The key observation is the following Lemma.

Lemma 8. *If $t \in \mathbb{L}_\mu$, then $nbr(t) \geq fb(\overline{\gamma(t)}) \geq \mathcal{E}(\overline{\gamma(t)})$. If $G \in \mathcal{G}$ and $Max^G = 0$, then there exists a μ-term $t \in \mathbb{L}_\mu$ such that $\gamma(t) \sim G$ and $nbr(t) = \mathcal{E}(\overline{G})$.*

We omit the proof for lack of space. Let us consider next the following classes:

$$\mathcal{L}_n^{\Sigma_1} = \{\, G \in \mathcal{G} \mid Max^G = 0 \text{ and } G \sim H \text{ for some } H \in \mathcal{G} \text{ s.t. } \mathcal{E}(\overline{H}) \leq n \,\}\,.$$

Let us suppose that $n \leq m$ and that $G \in \mathcal{G}$ is such that $Max^G = 0$, $\mathcal{E}(\overline{G}) = m$, and $G \notin \mathcal{L}_n^{\Sigma_1}$. Then we can find $t \in \mathbb{L}_\mu$ such that $nbr(t) = m$ and $\gamma(t) \sim G$. If $s \sim t$, then $\gamma(s) \sim G$ and hence $nbr(s) \geq \mathcal{E}(\overline{\gamma(s)}) > m$. That is, for such a $t \in \mathbb{L}_\mu$, we have $t \in \mathcal{T}_n \setminus \mathcal{T}_m$. Thus, separating the classes $\mathcal{L}_n^{\Sigma_1}$ implies separation of the classes $\mathcal{T}_n$.

The $\star$-weak simulation. We define in the following a relation between graphs, called *$\star$-weak simulation*, that shall be of use in comparing entanglements. Intuitively, there is a weak simulation of a graph G by H if every edge of G is simulated by a non empty finite path of H. Observe now that, in such a situation, if the simulating paths do not intersect, except that in their endpoints, then H contains a subgraph obtained from G by stretching edges into non empty paths. This property implies the relation $\mathcal{E}(G) = \mathcal{E}(H)$. However, for the weak simulations that arise when considering a game H which is semantically equivalent to a strongly synchronizing game G, see Section 4, only a weaker property holds: if the simulating paths do intersect, then the edges being simulated intersect in some of their endpoints. We call a weak simulation with this property a $\star$-weak simulation. The weaker property suffices to prove the comparison stated at the end of this Section, Theorem 15.

Definition 9. *A* weak simulation (R, ς) *of G by H is a binary relation $R \subseteq V_G \times V_H$ that comes with a partial function $\varsigma : V_G \times V_G \times V_H \longrightarrow \Pi^+(H)$, such that:*
– R is surjective, i.e. for every $g \in V_G$ there exists $h \in V_H$ such that gRh,
– R is functional, i.e. if $g_i R h$ for $i = 1, 2$, then $g_1 = g_2$,
– if gRh and $g \to g'$, then $\varsigma(g, g', h)$ is defined and $g' R h'$, where $h' = \delta_1 \varsigma(g, g', h)$.

Next we study conditions under which existence of a weak simulation of G by H implies that $\mathcal{E}(G)$ is some lower bound of $\mathcal{E}(H)$. To this goal, let us abuse of notation and write $h \in \varsigma(g, g', h_0)$ if $\varsigma(g, g', h_0) = h_0 h_1 \dots h_n$ and, for some $i \in \{0, \dots, n\}$, we have $h = h_i$. If $G = (V_G, E_G)$ is a directed graph then its undirected version $S(G) = (V_G, E_{S(G)})$ is the undirected graph such that $\{g, g'\} \in E_{S(G)}$ iff $(g, g') \in E_G$ or $(g', g) \in E_G$. We say that G has *girth at least* k if G does not contain loops, $(g, g') \in E_G$ implies $(g', g) \notin E_G$, and the shortest cycle in $S(G)$ has length at least k.

Definition 10. *We say that a weak simulation (R, ς) of G by H is a $\star$-weak simulation (or that it has the $\star$-property) if G has girth at least 4, and if $(g, g'), (\tilde{g}, \tilde{g}')$ are distinct edges of G and $h \in \varsigma(g, g', h_0), \varsigma(\tilde{g}, \tilde{g}', \tilde{h}_0)$, then $|\{\, g, g', \tilde{g}, \tilde{g}' \,\}| = 3$.*

We explain next this property. Given (R, ς), consider the set $C(h) = \{ (g, g') \in E_G \mid \exists h_0 \text{ s.t. } h \in \varsigma(g, g', h_0) \}$.

Lemma 11. *Let (R, ς) be a $\star$-weak simulation of G by H. If $C(h)$ is not empty, then there exists an element $c(h) \in V_G$ such that for each $(g, g') \in C(h)$ either $c(h) = g$ or $c(h) = g'$. If moreover $|C(h)| \geq 2$, then this element is unique.*

That is, $C(h)$ considered as an undirected graph, is a star. Since $c(h)$ is unique whenever $|C(h)| \geq 2$, then $c(h)$ is a partial function which is defined for all h with $|C(h)| \geq 2$. This allows to define a partial function $f : V_H \longrightarrow V_G$, which is defined for every h for which $C(h) \neq \emptyset$, as follows:

$$f(h) = \begin{cases} c(h), & |C(h)| \geq 2\,, \\ g, & \text{if } C(h) = \{ (g, g') \} \text{ and } h \text{ has no predecessor in } H\,, \\ g', & \text{if } C(h) = \{ (g, g') \} \text{ and } h \text{ has a predecessor in } H\,. \end{cases} \tag{2}$$

Let us remark that if $h \in \varsigma(g, g', h_0)$, then $f(h) \in \{ g, g' \}$. If gRh and h has no predecessor, then $f(h) = g$. Also, if h' is the target of $\varsigma(g, g', h_0)$ and g' has a successor, then $f(h') = g'$.

Lemma 12. *If (R, ς) is a $\star$-weak simulation of G by H and $\rho : K \longrightarrow H$ is an unravelling of H, then there exists a $\star$-weak simulation $(\tilde{R}, \tilde{\varsigma})$ of G by K.*

If H is a tree with back-edges, rooted at h_0, then we say that a winning strategy for Cops in the game $\mathcal{E}(H, k)$ from position $(h_0, \emptyset, Cops)$ is ***rigid*** if *every time Thief has to move from a position of form $(v, C, Thief)$ then for every back-edge $(v, u) \in B_H$ we have $u \in C$.*

Lemma 13. *Let H be a tree with back-edges, rooted at h_0, of feedback k, then Cops has a rigid winning strategy in $\mathcal{E}(H, k)$ from the position $(h_0, \emptyset, Cops)$.*

Remark 14. Let us remark that, by using a rigid strategy, (i) every path chosen by Thief in H is a tree path, (ii) if the position in $\mathcal{E}(H, k)$ is of the form $(h, C, Thief)$, and $h' \neq h$ is in the subtree of h, then the unique tree path from h to h' does contain no cops, apart possibly for the vertex h.

We establish next the connection between $\star$-weak simulations and entanglement.

Theorem 15. *If (R, ς) is a $\star$-weak simulation of G by H, then $\mathcal{E}(G) \leq \mathcal{E}(H)+2$.*

Proof. Let $k = \mathcal{E}(H)$. We shall define first a strategy for Cops in the game $\widetilde{\mathcal{E}}(G, k+2)$. In a second time, we shall prove that this strategy is a *winning* strategy for Cops.

Let us consider Thief's first move in $\widetilde{\mathcal{E}}(G, k+2)$. This move picks $g \in G$ leading to the position $(g, \emptyset, Cops)$ of $\widetilde{\mathcal{E}}(G, k+2)$. Cops answers by occupying the current position, i.e. he moves to $(g, \{ g \}, Thief)$. After this move, Cops also chooses a tree with back-edges of feedback k to which H unravel, $\pi : \mathcal{T}(H) \longrightarrow H$, such that the root h_0 of $\mathcal{T}(H)$ satisfies $gR\pi(h_0)$. We can also suppose that h_0 is not

a return, thus it has no predecessor. According to Lemma 12 we can lift the $\star$-weak simulation (R, ς) to a $\star$-weak simulation $(\tilde{R}, \tilde{\varsigma})$ of G by $\mathcal{T}(H)$. In other words, we can suppose from now on that H itself is a tree with back-edges of feedback k rooted at h_0 and, moreover, that gRh_0.

From this point on, Cops uses a memory to choose how to place cops in the game $\widetilde{\mathcal{E}}(G, k+2)$. To each Thief's position $(g, C_G, Thief)$ in $\widetilde{\mathcal{E}}(G, k+2)$ we associate a data structure (the memory) consisting of a triple $M(g, C, Thief) = (p, c, h)$, where $c, h \in V_H$ and $p \in V_H \cup \{\perp\}$ (we assume that $\perp \notin V_H$). Moreover c is an ancestor of h in the tree and, if $p \neq \perp$, then p is an ancestor of c as well.

Intuitively, we are matching the play in $\widetilde{\mathcal{E}}(G, k+2)$ with a play in $\mathcal{E}(H, k)$, started at the root h_0 and played by Cops according to a rigid strategy, Lemma 13. Thus c is the vertex of H currently occupied by Thief in the game $\mathcal{E}(H, k)$.[2] Instead of recalling all the play (that is, the history of all the positions played so far), we need to record the last position played in $\mathcal{E}(H, k)$: this is p, which is undefined when the play begins. Cops on G are positioned on the images of Cops on H by the function f defined in (2). Moreover, Cops eagerly occupies the last two vertices visited on G. Thief's moves on G are going to be simulated by sequences of Thief's moves on H, using the $\star$-weak simulation (R, ς). In order to make this possible, a simulation of the form $\varsigma(\tilde{g}, g, \tilde{h})$ must be halted before its target h; the current position c is such halt-point. This implies that the simulation of $g \to g'$ by (R, ς) and the sequence of moves in H matching Thief's move on G are sligthly out of phase. To cope with that, Cops must guess in advance what might happen in the rest of the simulation and this is why he puts cops on the current and previous positions in G. We also need to record h, the target of the previous simulation into the memory.

The previous considerations are formalized by requiring the following conditions to hold. To make sense of them, let us say that $f(\{p\}) = f(p)$ if $p \in V_H$ and that $f(\{p\}) = \emptyset$ if $p = \perp$. In the last two conditions we require that $p \neq \perp$.

- $C_G = f(C_H(c)) \cup f(\{p\}) \cup \{g\}$, (COPS)
- $f(c) = g$, and $f(h') \in f(\{p\}) \cup \{g\}$, if h' lies on the tree path from c to h, (TAIL)
- $f(p) \to g$, $f(p) R \tilde{h}$ for some $\tilde{h} \in V_H$, $c \in \varsigma(f(p), g, \tilde{h})$, and h is the target of $\varsigma(f(p), g, \tilde{h})$, (HEAD)
- on the tree path from p to c, c is the only vertex s.t. $f(c) = g$. (HALT)

Since h_0 has no predecessors, then gRh_0 implies $f(h_0) = g$. Thus, at the beginning, the memory is set to $(\perp, h_0, h_0)$ and conditions (COPS) and (TAIL) hold.

Consider now a Thief's move of the form $(g, C_G, Thief) \to (g', C_G, Cops)$, where $g' \notin C_G$. If g' has no successor, then Cops simply skips, thus reaching a

[2] More precisely we are associating to the position $(g, C_G, Thief)$ of $\mathcal{E}(G, k+2)$ a position $(c, C_H, Thief)$ in $\mathcal{E}(H, k)$, where C_H is a superset of the set of returns that are active in c, see Remark 14.

winning position. Let us assume that g' has a successor, and write $\varsigma(g, g', h) = hh_1 \dots h_n$, $n \geq 1$; observe that $f(h_n) = g'$. If for some $i = 1, \dots, n$ h_i is not in the subtree of c, then the strategy halts, Cops abandons the game and looses. Otherwise, all the path $\pi = c \dots hh_1 \dots h_n$ lies in the subtree of c. By eliminating cycles from π, we obtain a simple path σ, of source c and target h_n, which entirely lies in the subtree of c. By Lemma 1, σ is the tree path from c to h_n. An explicit description of σ is as follows: we can write σ as the compose $\sigma_0 \star \sigma_1$, where the target of σ_0 and source of σ_1 is the vertex of $\varsigma(g, g', h)$ which is closest to the root h_0; moreover σ_0 is a prefix of the tree path from c to h, and σ_1 is a postfix of the path $\varsigma(g, g', h)$.

We cut σ as follows: we let c' be the first vertex on this path such that $f(c') = g'$. Thief's move $g \to g'$ on G is therefore simulated by Thief's moves from c to c' on H. This is possible since every vertex lies in the subtree of c and thus it has not yet been explored. Cops consequently occupies the returns on this path, thus modifying C_H to $C'_H = C_H(c') = (C_H \setminus X) \uplus Y$, where $X \subseteq C_H$ and Y is a set of at most k vertexes containing the last returns visited on the path from c to c'.

After the simulation on H, Cops moves to $(g', C'_G, Thief)$ in $\widetilde{\mathcal{E}}(G, k+2)$, where $C'_G = f(C'_H) \cup \{ g, g' \}$. Let us verify that this is an allowed move according to the rules of the game. We remark that $f(Y) \subseteq f(\{ p \}) \cup \{ g, g' \}$ and therefore

$$\begin{aligned} C'_G &= f(C_H \setminus X) \cup f(Y) \cup \{ g, g' \} \\ &= (f(C_H \setminus X) \cup (f(Y) \setminus \{ g' \}) \cup \{ g \}) \cup \{ g' \} = A \cup \{ g' \}, \end{aligned}$$

where $A = f(C_H \setminus X) \cup (f(Y) \setminus \{ g' \}) \cup \{ g \} \subseteq f(C_H) \cup f(\{ p \}) \cup \{ g \} = C_G$. After the simulation Cops also updates the memory to $M(g', C'_G, Thief) = (c, c', h_n)$. Since $f(c) = g$, then condition (COPS) clearly holds. Also, $f(c) = g \to g'$, gRh and h_n is the target of $\varsigma(f(c), g', h)$. We have also that $c' \in \sigma_1$ and hence $c' \in \varsigma(f(c), g', h)$, since otherwise $c' \in \sigma_0$ and $f(c') \in \{ f(p), g \}$, contradicting $f(c') = g'$ and the condition on the girth of G. Thus condition (HEAD) holds as well. Also, condition (HALT) holds, since by construction c' is the first vertex on the tree path from c to h such that $f(c') = g'$. Let us verify that condition (TAIL) holds: by construction $f(c') = g'$, and the path from c' to h_n is a postfix of $\varsigma(g, g', h)$, and hence $f(h') \in \{ g, g' \}$ if h' lies on this tree path.

Let us now prove that the strategy is winning. If Cops never abandons, then an infinite play in $\widetilde{\mathcal{E}}(G, k+2)$ would give rise to an infinite play in $\mathcal{E}(H, k)$, a contradiction. Thus, let us prove that Cops will never abandon. To this goal we need to argue that when Thief plays the move $g \to g'$ on G, then the simulation $\varsigma(g, g', h) = hh_1 \dots h_n$ lies in the subtree of c. If this is not the case, let i be the first index such that h_i is not in the subtree of c. Therefore h_i is a return and, by the assumptions on H and on rigid strategy, $h_i \in C_H(c)$. Since $h_i \in \varsigma(g, g', h)$, $f(h_i) \in \{ g, g' \}$. Observe, however that we cannot have $f(h_i) = g'$, otherwise $g' \in f(C_H(c)) \subseteq C_G$. We deduce that $f(h_i) = g$ and that $g \in f(C_H) \subseteq C_G$.

Since $C_G \neq \bot$, then $(g, C_G, Thief)$ is not the initial position of the play, so that, if $M(g, C_G, Thief) = (p, c, h)$, then $p \neq \bot$. Let us now consider the last two moves of the play before reaching position $(g, C_G, Thief)$. These are of the

form $(f(p), \tilde{C}_G, Thief) \to (g, \tilde{C}_G, Cops) \to (g, C_G, Thief)$, and have been played according to this strategy. Since $g \notin \tilde{C}_G$, it follows that the Cop on h_i has been dropped on H during the previous round of the strategy, simulating the move $f(p) \to g$ on G by the tree path from p to c. This is however in contradiction with condition (HALT), stating that c is the only vertex h on the tree path from p to c such that $f(h) = c$. □

4 Strongly Synchronizing Games

In this section we define *strongly synchronizing* games, a generalization of synchronizing games introduced in [5]. We shall show that, for every game H equivalent to a strongly synchronizing game G, there is a $\star$-weak simulation of G by H. Let us say that $G \in \mathcal{G}$ is *bipartite* if $M^G \subseteq Pos_E^G \times Pos_{A,D}^G \cup Pos_A^G \times Pos_{E,D}^G$.

Definition 16. *A game G is* strongly synchronizing *if it is bipartite, it has girth strictly greater than 4 and, for every pair of positions g, k, the following conditions hold:*

1. *if $(G, g) \sim (G, k)$ then $g = k$.*
2. *if $(G, g) \leq (G, k)$ and $(G, k) \not\leq (G, g)$, then $k \in Pos_E^G$ and $(k, g) \in M^G$, or $g \in Pos_A^G$ and $(g, k) \in M^G$.*

A consequence of the previous definition is that *the only winning strategy for Mediator in the game $\langle G, G\rangle$ is the copycat strategy.* Thus strongly synchronizing games are synchronizing as defined in [5]. We list next some useful properties of strongly synchronizing games.

Lemma 17. *Let G be a strongly synchronizing and let $(g, g'), (\tilde{g}, \tilde{g}') \in M^G$ be distinct.*

1. *If $(G, g) \sim \hat{x}$ then $g \in Pos_D^G$ and $\lambda(g) = x$.*
2. *If $g, \tilde{g} \in Pos_E^G$ and, for some game H and $h \in Pos^H$, we have $(G, g') \leq (H, h) \leq (G, g)$ and $(G, \tilde{g}') \leq (H, h) \leq (G, \tilde{g})$, then $g = \tilde{g}$ or $g' = \tilde{g}'$, and $|\{ g, g', \tilde{g}, \tilde{g}' \}| = 3$.*
3. *If $g \in Pos_E^G$ and $\tilde{g} \in Pos_A^G$ and, for some H and $h \in Pos^H$, we have $(G, g') \leq (H, h) \leq (G, g)$ and $(G, \tilde{g}) \leq (H, h) \leq (G, \tilde{g}')$, then $g = \tilde{g}'$ or $g' = \tilde{g}$, and $|\{ g, g', \tilde{g}, \tilde{g}' \}| = 3$.*

We are ready to state the main result of this section.

Proposition 18. *Let G be a strongly synchronizing game, and let $H \in \mathcal{G}$ be such that $G \leq H \leq G$, then there is a $\star$-weak simulation of G by H.*

Proof. Let S, S' be two winning strategies for Mediator in $\langle G, H\rangle$ and $\langle H, G\rangle$, respectively. Let $T = S||S'$ be the composal strategy in $\langle G, H, G\rangle$. Say that gRh if (g, h, g) is a position of T and g, h belong to the same player. We consider first R and prove that it is functional and surjective. If $g_i R h, i = 1, 2$ then (g_1, h, g_1) and (g_2, h, g_2) are positions of T, hence $(G, g_1) \leq (H, h) \leq (G, g_1)$ and $(G, g_2) \leq (H, h) \leq (G, g_2)$, consequently $(G, g_1) \sim (G, g_2)$ implies $g_1 = g_2$,

by definition 16. For surjectivity, we can assume that (a) all the positions of G are reachable from the initial position $p_\star^G$, (b) $p_\star^G$ and $p_\star^H$ belong to the same player (by possibly adding to H a new initial position leading to the old one). Since $T_{\backslash H}$ is the copycat strategy, given $g \in Pos^G_{E,A,D}$, from the initial position $(p_\star^G, p_\star^H, p_\star^G)$ of $\langle G, H, G\rangle$, the Opponents have the ability to reach a position of the form (g, h, g). The explicit construction of the function ς will show that h can be chosen to belong to the same player as g.

We construct now the function ς so that (R, ς) is a weak simulation. If gRh and $(g, g') \in M^G$, then we construct $\pi = h, \ldots, h'$ such that $g'Rh'$. Since G is bipartite, then $h \neq h'$ and π is nonempty. We let $\varsigma(g, g', h)$ be a reduction of π to a nonempty simple path.

We assume $(g, h) \in (Pos^G_E, Pos^H_E)$, the case $(g, h) \in (Pos^G_A, Pos^H_A)$ is dual. From position (g, h, g) it is Opponent's turn to move on the left, they choose a move $(g, g') \in M^G$. Since G is bipartite, we have either $g' \in Pos^G_D$ or $g' \in Pos^G_A$.
Case (i). If $g' \in Pos^G_D$ then the strategy T suggests playing a finite path on H, $(g', h, g) \rightarrow^* (g', h^*, g)$, possibly of zero length, and then it will suggest to play on the external right board. An infinite path played only on H cannot arise, since T is a winning strategy and such an infinite path is not a win for Mediator. Since $T_{\backslash H}$ is the copycat strategy, T suggests the only move $(g', h^*, g) \rightarrow (g', h^*, g')$. From this position T suggests playing a path on H leading to a final draw position $h_f \in Pos^H_D$ as follows $(g', h^*, g') \rightarrow^* (g', h_f, g')$, such that $\lambda^G(g') = \lambda^H(h_f)$, therefore $g'Rh_f$.
Case (ii). If $g' \in Pos^G_A$ then from position (g', h, g) it is Mediator's turn to move. We claim that T will suggest playing a nonempty finite path $(g', h, g) \rightarrow^+ (g', h', g)$ on the central board H, where $h' \in Pos^H_A$, and then suggests the move $(g', h', g) \rightarrow (g', h', g')$. Let $\tilde{h} \in Pos^H_{A,E,D}$ be such that the position $(g', \tilde{h}, g)$ has been reached from (g', h, g), through a (possibly empty) sequence of central moves, by playing with T. Then T cannot suggest a move on the left board $(g', \tilde{h}, g) \rightarrow (g'', \tilde{h}, g)$, since $T_{\backslash H}$ is the copycat strategy. Also, if $\tilde{h} \in Pos^H_E$, T cannot suggest a move on the right board $(g', \tilde{h}, g) \rightarrow (g', \tilde{h}, \tilde{g})$. The reason is that $T = S||S'$, and the position $(\tilde{h}, g)$ of $\langle H, G\rangle$ does not allow a Mediator's move on the right board. Thus a sequence of central moves on H is suggested by T and, as mentioned above, this sequence cannot be infinite. We claim that its endpoint $h' \in Pos^H_A$. We already argued that $h' \notin Pos^H_E$, let us argue that $h' \notin Pos^H_D$. If this were the case, then strategy T suggests the only move $(g', h', g) \rightarrow (g', h_n, g')$, hence $(G, g') \sim (H, h')$. By Lemma 17.1, we get $g' \in Pos^G_D$, contradicting $g' \in Pos^G_A$.

This proves that (R, ς) is a weak simulation. We prove next that (R, ς) has the $\star$-property, thus assume that $h^* \in \varsigma(g, g', h_0), \varsigma(\tilde{g}, \tilde{g}', \tilde{h}_0)$. Let us suppose first that $g, \tilde{g} \in Pos^H_E$. By looking at the construction of these paths, we observe that the two sequences of moves

$$(g, h_0, g) \rightarrow (g', h_0, g) \rightarrow^* (g', h^*, g) \rightarrow^* (g', h_n, g) \rightarrow (g', h_n, g'),$$
$$(\tilde{g}, \tilde{h}_0, \tilde{g}) \rightarrow (\tilde{g}', \tilde{h}_0, \tilde{g}) \rightarrow^* (\tilde{g}', h^*, \tilde{g}) \rightarrow^* (\tilde{g}', \tilde{h}_m, \tilde{g}) \rightarrow (\tilde{g}', \tilde{h}_m, \tilde{g}'),$$

may be played in the game $\langle G, H, G\rangle$, according to the winning strategy $T = S||S'$. We have therefore that $(G, g') \leq (H, h^*) \leq (G, g)$ and $(G, \tilde{g}') \leq (H, h^*) \leq (G, \tilde{g})$.[3] Consequently $|\{\, g, g', \tilde{g}, \tilde{g}' \,\}| = 3$, by Lemma 17.2. If $g \in Pos_E^G$ and $\tilde{g} \in Pos_A^G$, a similar argument shows that the positions (g', h^*, g) and $(\tilde{g}, h^*, \tilde{g}')$ may be reached with T and hence $(G, g') \leq (H, h^*) \leq (G, g)$ and $(G, \tilde{g}) \leq (H, h^*) \leq (G, \tilde{g}')$. Lemma 17.3 implies that $|\{\, g, g', \tilde{g}, \tilde{g}' \,\}| = 3$. Finally, the cases $(g, \tilde{g}) \in \{\, (Pos_A^G, Pos_A^G), (Pos_A^G, Pos_E^G) \,\}$ are handled by duality. This completes the proof of Proposition 18. □

5 Construction of Strongly Synchronizing Games

In this section we complete the hierarchy theorem by constructing, for $n \geq 1$, strongly synchronizing games G_n such that $\mathcal{E}(G_n) = n$. The game G_2 appears in Figure 1. The general definition of the game G_n is as follows. Let $[n]$ denote the

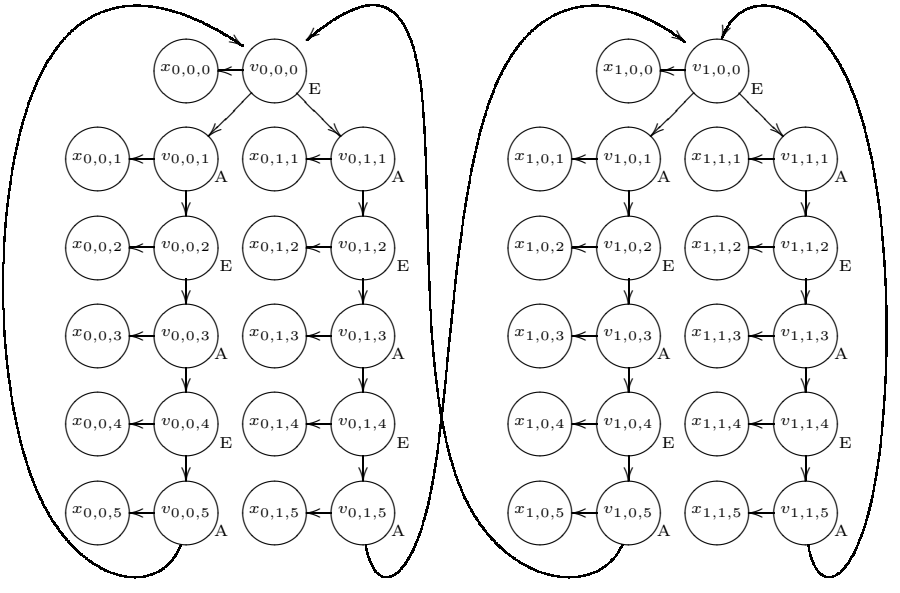


Fig. 1. The game G_2

set $\{\, 0, \ldots, n-1 \,\}$ and let $I_n = \{\, (i,j,k) \in [n] \times [n] \times [6] \mid k = 0 \text{ implies } j = 0 \,\}$. For $P \in \{\, E, P \,\}$ and $x_P \in \{\, 0, 1 \,\}$ with $x_P = 0$ iff $P = E$, let

$$Pos_P^{G_n} = \{\, v_{i,j,k} \mid (i,j,k) \in I_n \text{ and } k \bmod 2 = x_P \,\}, \quad Pos_D^{G_n} = \{\, w_y \mid y \in I_n \,\}.$$

Let $X = \{\, x_{i,j,k} \mid i, j \geq 0, k \in [n] \,\}$ be a countable set of variables, the labelling of draw positions, $\lambda^{G_n} : Pos_D^{G_n} \longrightarrow X$, sends $w_{i,j,k}$ to $x_{i,j,k}$. The moves M^{G_n} either

[3] Similar inequalites may be derived even if $h^* \in Pos_D^H$. In this case the moves in the central board may be interleaved with the move on the right board.

lie on some cycle, $v_{i,0,0} \to v_{i,j,1}$, $v_{i,j,k} \to v_{i,j,k+1}$, $k = 1, \ldots, 4$, $v_{i,j,5} \to v_{j,0,0}$ or lead to draw positions, $v_{i,j,k} \to w_{i,j,k}$. Finally, the priority function ρ^{G_n} assigns a zero priority to all positions.

Proposition 19. *The games G_n are strongly synchronizing and $\mathcal{E}(G_n) = n$.*

We are now ready to state the main achievement of this paper.

Theorem 20. *For $n \geq 3$, the inclusions $\mathcal{L}_{n-3} \subseteq \mathcal{L}_n$ are strict. Therefore the variable hierarchy for the lattice μ-calculus is infinite.*

By the previous Proposition the game $G_n \in \mathcal{L}_n$. Also, since G_n is strongly synchronizing, if $H \sim G_n$, then there exists a $\star$-weak simulation of G_n by H. It follows by Theorem 15 that $n - 2 \leq \mathcal{E}(H)$. Therefore $G_n \notin \mathcal{L}_{n-3}$.

References

1. Bloom, S.L., Ésik, Z.: Iteration theories. EATCS Monographs on Theoretical Computer Science. Springer, Berlin (1993)
2. Arnold, A., Niwiński, D.: Rudiments of μ-calculus. Studies in Logic and the Foundations of Mathematics, vol. 146. North-Holland, Amsterdam (2001)
3. Bradfield, J.C.: The modal mu-calculus alternation hierarchy is strict. Theor. Comput. Sci. 195(2), 133–153 (1998)
4. Arnold, A.: The μ-calculus alternation-depth hierarchy is strict on binary trees. Theor. Inform. Appl. 33(4-5), 329–339 (1999)
5. Santocanale, L.: The alternation hierarchy for the theory of μ-lattices. Theory Appl. Categ. 9, 166–197 (2002)
6. Santocanale, L., Arnold, A.: Ambiguous classes in μ-calculi hierarchies. Theor. Comput. Sci. 333(1-2), 265–296 (2005)
7. Eggan, L.C.: Transition graphs and the star-height of regular events. Mich. Math. J. 10, 385–397 (1963)
8. Braquelaire, J.P., Courcelle, B.: The solutions of two star-height problems for regular trees. Theor. Comput. Sci. 30(2), 205–239 (1984)
9. Berwanger, D., Grädel, E., Lenzi, G.: The variable hierarchy of the μ-calculus is strict. Theory Comput. Syst. 40(4), 437–466 (2007)
10. Blass, A.: A game semantics for linear logic. Ann. Pure Appl. Logic 56(1-3), 183–220 (1992)
11. Nerode, A., Yakhnis, A., Yakhnis, V.: Concurrent programs as strategies in games. In: Moschovakis, Y.N. (ed.) Logic from Computer Science, pp. 405–479. Springer, Heidelberg (1992)
12. Abramsky, S., Jagadeesan, R.: Games and full completeness for multiplicative linear logic. J. Symb. Logic 59(2), 543–574 (1994)
13. Joyal, A.: Free lattices, communication and money games. In: Logic and scientific methods. Synthese Lib, vol. 259, pp. 29–68. Kluwer Acad. Publ., Dordrecht (1997)
14. Freese, R.: Free lattices. Math. Surveys and Monographs. vol. 42, AMS (1995)
15. Joyal, A.: Free bicomplete categories. C. R. Math. Canada 17(5), 219–224 (1995)
16. Cockett, J.R.B., et al.: Finite sum-product logic. Theory Appl. Categ. 8, 63–99 (2001)
17. Santocanale, L.: Free μ-lattices. Jour. of Pure and Applied Algebra 168(2-3), 227–264 (2002)

18. Santocanale, L.: A calculus of circular proofs and its categorical semantics. In: Nielsen, M., Engberg, U. (eds.) FOSSACS 2002. LNCS, vol. 2303, pp. 357–371. Springer, Heidelberg (2002)
19. Arnold, A., Santocanale, L.: Ambiguous classes in the games mgr-calculus hierarchy. In: Gordon, A.D. (ed.) FOSSACS 2003. LNCS, vol. 2620, pp. 70–86. Springer, Heidelberg (2003)
20. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pacific J. Math. 5, 285–309 (1955)
21. Berwanger, D., Grädel, E.: Entanglement – A measure for the complexity of directed graphs with applications to logic and games. In: Baader, F., Voronkov, A. (eds.) LPAR 2004. LNCS, vol. 3452, pp. 209–223. Springer, Heidelberg (2005)

A Formalised Lower Bound on Undirected Graph Reachability

Ulrich Schöpp

Insititut für Informatik
Ludwig-Maximilians-Universität München
Oettingenstraße 67, D-80538 München, Germany

Abstract. We study the expressivity of Jumping Automata on Graphs (JAGs), an idealised model of computation with logarithmic space. JAGs operate on graphs by using finite control and a constant number of pebbles. In this paper we revisit the proof of Cook & Rackoff that JAGs cannot decide s-t-reachability in undirected graphs. Cook & Rackoff prove this result by constructing, for any given JAG, a finite graph that cannot be traversed exhaustively by the JAG. We generalise this result from the graphs constructed by Cook & Rackoff to a general class of group action graphs. We establish a bound on the number of nodes that a JAG can visit on such action graphs. This generalisation allows us to strengthen the result of Cook & Rackoff to the existence of a graph of small degree whose diameter (rather than its number of nodes) is larger than the number of nodes the JAG can visit. The main result has been formalised in the theorem prover Coq, using Gonthier's tactic language SSReflect.

1 Introduction

Many LOGSPACE-algorithms can be viewed as taking some structured input, e.g. a graph, on which they operate by means of a constant number of pointers, e.g. to the nodes of the graph. Describing LOGSPACE-algorithms as such pointer programs is not only usually easier than working directly with Turing Machines, it is also useful for studying the nature of computation with logarithmic space. With current techniques only very few results can be proven about what cannot be done in logarithmic space. Computation models of idealised pointer algorithms are more accessible and may serve as a starting point for obtaining insight into the general problem.

A classic example of a class of pointer algorithms introduced for this purpose are the *Jumping Automata on Graphs* (JAGs) of Cook & Rackoff [1]. JAGs operate on graphs with a local ordering, which means that the outgoing edges of each node are numbered consecutively starting from 1. A JAG is an automaton with finite control that can place a constant number of pebbles on the input graph. In each step the automaton may observe the incidence relation of the pebbles, i.e. it can tell which of its pebbles happen to lie on the same graph node. Using this information it may then change its state and move a pebble of its choice along a graph edge (identified by a number) or jump it to the location of another pebble.

With these operations, JAGs capture a minimal core of what one would expect pointer programs to be able to do. Although JAGs do not capture all reasonable pointer

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 621–635, 2008.

programs, a good knowledge of their properties should be helpful in analysing the expressivity of extensions that capture larger classes of pointer programs. For example, JAGs lack the ability to decide if all nodes of the input graph have some property, since they cannot reach an isolated component of the input graph if the start configuration does not place a pebble on it. One way of removing this unnatural limitation is to study extensions of JAGs with universal iteration, where it is possible for the machine to iterate with one pebble over all graph nodes. Deterministic Transitive Closure (DTC) logic on locally ordered graphs features first-order quantifiers and may be viewed as an extension of JAGs of this kind. Since the computation of an extended JAG with universal iteration amounts to a sequence of ordinary JAG-computations interrupted by iteration-jumps, we believe that a good knowledge of ordinary JAG-computations should be helpful in analysing the expressivity of the extended JAG-model.

One interesting problem to consider when studying the expressivity of idealised computation models for deterministic LOGSPACE is that of s-t-reachability in undirected graphs. Reingold has recently shown that this problem can be solved in logarithmic space [10]. His algorithm may be expressed as a pointer program that has access to a constant number of counting registers of logarithmic size [9]. However, many models of pointer programs that have been studied before do not feature such counting registers and are not able to encode them by pointer arithmetic. Examples of such models include the above-mentioned JAGs and DTC-logic on locally ordered graphs. Indeed, it has been shown by Cook & Rackoff [1] that JAGs cannot decide undirected reachability. For DTC-logic it is not known if undirected reachability for locally ordered graphs can be expressed, but Etessami & Immerman [2] have shown that the result of Cook & Rackoff can be used to obtain a partial negative result. This evidence raises the interesting question if counting registers are necessary for pointer programs to decide s-t-reachability in undirected graphs.

The study of this question has led us to revisit the proof of Cook & Rackoff that JAGs cannot decide reachability in undirected graphs. Cook & Rackoff obtain this result by constructing, for any JAG J, a graph that has more nodes than J can visit. In this paper we show that this proof can be generalised to yield the stronger result that there exists a graph whose *diameter* is larger than the number of nodes that J can visit. We formulate this result in a general way, so that it can be used for other graphs, without having to adapt the proof again. We expect the strengthening of Cook & Rackoff's result to be useful for analysing the expressivity of a JAG-model with universal iteration.

One of the main contributions of this paper is the formalisation of the main result in the theorem prover Coq, thus giving the highest confidence in the correctness of Cook & Rackoff's proof and the generalisation we make in this paper. We discuss the formalisation after presenting the proof that has been formalised.

2 Outline

We start with an informal outline of Cook & Rackoff's proof leading us to the generalisations in this paper.

Cook & Rackoff study the behaviour of JAGs on the d-dimensional torus of side-length m. The nodes of this graph are d-tuples of numbers from 0 to $m-1$. To define

the edges, define an addition on nodes by pointwise addition modulo m and for each $i \in \{1, \ldots, d\}$ define g_i to be the vector having the number 1 in the i-th component and 0 elsewhere. Then, for each i, there is an undirected edge between x and $x + g_i$.

The reason for studying these graphs is their cyclicity in all directions. If the pebble-moves of a JAG become periodic after a certain amount of time then, by the cyclic nature of the graph, there will be a repeat of pebble configurations not long after. The aim is now to show that for any JAG J we can choose m and d large enough, so that a repeat of configurations appears before J has had the time to visit all graph nodes. The key to doing this is to derive a good upper bound on the time the moves of J become periodic.

A bound on the time the pebble moves of a JAG J become periodic can be obtained by showing that J can only behave in a limited number of ways. The behaviour of J from a certain starting configuration is the sequence of moves it makes together with the states it assumes. If any computation is longer than the number of possible behaviours then two configurations from which J behaves the same must appear in the sequence, so that the pebble moves between those configurations will be repeated periodically.

To analyse in how many different ways a JAG may behave, it is useful to introduce a notion of extended state. Since we must analyse the behaviour of any JAG, we do not know, in general, what information the JAG stores in its state. To be able to say something about all machines, we study what information a machine could maximally obtain about the graph and the position of the pebbles. To see what a JAG could learn about the torus, notice first that the neighbourhoods of all nodes in this graph are isomorphic. Hence, the only thing a JAG J can learn about this graph is the relative position of pebbles. At the start of the computation J knows only which pebbles lie on the same node, but otherwise has no information about their relative positions. During the course of the computation, the JAG could now (if it had enough states) keep track of the relative distance of any two pebbles that did coincide on the same node at some point earlier in the computation. For any two pebbles the relative distance can be described as a d-vector z, such that if x and y are the positions of the pebbles then $x = y + z$ holds. In each step of the JAG, the known pebble distances can be updated according to the move, for example by adding g_i in the case of a pebble move along edge i.

We call the *extended state* the state of the JAG together with the maximal information of relative pebble distances recorded as just outlined. An important property of extended states is that many pebble-collisions can be predicted from it. If the distance of two pebbles is known in an extended state, then one can tell by looking at the extended state whether or not they will collide in the next computation step. Unexpected collisions may only happen in edge moves and only between pebbles whose distance is not already known. After such a collision the distance of the colliding pebble becomes known. Cook & Rackoff call this event a *coalition*, since if we consider the equivalence relation on pebbles of "having known distance," two equivalence classes are united at this point. If P denotes the set of pebbles of the JAG then in each computation sequence there can appear only $|P|$ coalitions, as an equivalence relation on P can have at most $|P|$ equivalence classes. Notice furthermore that in steps with a coalition, the extended state after the step is determined by the previous extended state together with the information which pebbles collided unexpectedly. In all other steps, the new extended state after the step is completely determined by the extended state before the step.

The bounds on the number of possible different behaviours of a JAG can now be established by induction on the number of coalitions. In computations without any coalition, the behaviour of the JAG is uniquely determined by the extended state at the start of the computation. A bound can then be given easily. It depends only on the number of states and pebbles of the JAG. For the induction step, consider the behaviour of a JAG in computation sequences with up to $k+1$ coalitions. By induction hypothesis, we know how many behaviours there are in computations with at most k coalitions. Now it is not hard to show that a JAG behaves the same in two computations with $(k+1)$ coalitions if in both computations it behaves the same before the $(k+1)$-th coalition, if the $(k+1)$-th coalition appears at the same time in both computations, and if the same pebbles collide in these two coalitions. Because of this observation and the induction hypothesis, we can give a bound on the number of possible behaviours of the JAG, provided we can find an upper bound on the time when a $(k+1)$-th coalition occurs. This can be done as follows. A $(k+1)$-th coalition can only appear if the computation has not gone into a loop before reaching it. Using the cyclicity of the torus and the induction hypothesis, we can get a bound on the number of different configurations that may appear in any computation sequence with up to k coalitions. This bound may be used to find an upper bound on the length of computation sequences that have neither reached a loop nor a $(k+1)$-th coalition. This then allows one to give an upper bound on the time of a $(k+1)$-th coalition and therefore can be used to give a bound on the number of possible behaviours of the JAG in computations with at most $(k+1)$ coalitions.

The result one obtains from this estimation is that there exists a constant c such that any JAG with $|P|$ pebbles and $|Q|$ states can visit at most $(|Q|m)^{c^{|P|}}$ nodes in the d-dimensional torus of side-length m. If we choose $d > 2c^{|P|}$ and $m = |Q|$ then any such machine can visit at most $(|Q|m)^{c^{|P|}} = m^{2c^{|P|}} < m^d$ nodes. Since the torus has m^d nodes, the machine can therefore not traverse it exhaustively.

Examination of this proof shows that only a few properties of the d-dimensional torus of side-length m are used in it: (i) the neighbourhoods of all nodes are isomorphic; (ii) by recording the distance of two pebbles, one can predict when two pebbles with known distance will collide; and (iii) the graph has exponent m, meaning that if we repeat the same edge moves m times then we return to the place we started from.

Based on this observation, we generalise the proof of Cook & Rackoff to a class of *action graphs* that have these three properties. We show that this generalisation allows us to strengthen the result of Cook & Rackoff to the existence of a graph in which the JAG can visit less nodes than the diameter of the graph, rather than its number of nodes.

3 Action Graphs

Action graphs are given by a node set with a group action. We write groups multiplicatively and write e_D or just e for the unit element of group D. The *exponent* of a group D is the smallest number m such that $x^m = e_D$ holds for all $x \in D$. An *action* $\odot$ of a group D on a set V is a function $\odot\colon V \times D \to V$ satisfying $v \odot e_D = v$ and $v \odot (x \cdot y) = (v \odot x) \odot y$ for all $v \in V$ and $x, y \in D$. A group action $\odot$ is *free* if $v \odot x = v$ implies $x = e_D$ for all $v \in V$ and $x \in D$.

Definition 1. *Let V be a finite set of nodes, D be a finite group with elements $g_1, \ldots, g_d$, and $\odot$ be a group action of D on V. Then the* action graph $G(V, D, \vec{g}, \odot)$ *is the locally ordered graph with node set V, in which the i-th edge from v goes to $v \odot g_i$ for all $v \in V$ and $i \in \{1, \ldots, d\}$. This action graph is a* free action graph *if the group action is free. We say that this* graph has exponent m *if the group D does.*

We work with free action graphs in order to satisfy requirements (i) and (ii) above. In such graphs any two nodes look the same, i.e. have the same isomorphism type, so that (i) is satisfied. For (ii) we note that we can use group elements to keep track of pebble distances: a group element x represents the distance of pebble locations v and w if $w = v \odot x$ holds. We can then predict pebble collisions from known distances, since if x represents the distance of v and w then we have $v = w$ if and only if $x = e$. Finally, to satisfy (iii) we will consider action graphs of (small) exponent m.

An important example of free action graphs are Cayley graphs. The Cayley graph $C(D, \vec{g})$ of the group D with respect to the group elements $\vec{g}$ is the free action graph $G(D, D, \vec{g}, \cdot)$, where $\cdot$ is the group multiplication. Any free action graph $G(V, D, \vec{g}, \odot)$ is in fact the disjoint union of one or more copies of the Cayley graph $C(D, \vec{g})$.

Example 2. For all $m, d > 0$, the d-dimensional torus of side-length m is the Cayley graph $G_{m,d} = C(D_{m,d}, g_1, \ldots, g_{2d})$, where $D_{m,d}$ is the commutative group $(\mathbb{Z}/m\mathbb{Z})^d$ of d-tuples of the cyclic group of order m with pointwise addition, and the vectors $g_1, \ldots, g_{2d}$ are defined such that for any $i \in \{1, \ldots, d\}$ we have $g_i = -g_{i+d}$ and the tuple g_i has a 1 in the i-th position and 0s elsewhere. We use additive notation for the group $D_{m,d}$.

The graph $G_{m,d}$ is undirected, since for any edge from v to $v + g_i$ there is also an edge from $v + g_i$ to $v + g_i - g_i = v$. This graph $G_{m,d}$ has degree $2d$ and exponent m. It has m^d nodes and its diameter is at most md. □

Having defined action graphs, we can now state the main result we aim to show in this paper in the following proposition.

Proposition 3. *There exists a constant c, such that, for any free action graph G with exponent m and any* JAG *J with $|Q|$ states and $|P|$ pebbles, the number of nodes that J can visit from any start configuration on G is at most $(m|Q|)^{c^{|P|}}$.*

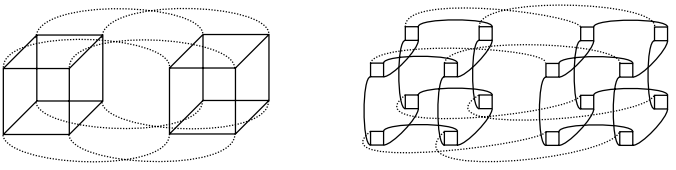
We prove this proposition in the next section. In the rest of the present section, we discuss how it generalises the result of Cook & Rackoff by instantiating it with different free action graphs.

With the family of graphs $G_{m,d}$, the proposition is just the result of Cook & Rackoff. If $d > 2c^{|P|}$ and $m = |Q|$ then no JAG J with $|P|$ pebbles and $|Q|$ states can visit all nodes in $G_{m,d}$. We note that the degree $2d$ depends only on the number of pebbles and not on the number of states. This is important, for example for proving the corollary that no JAG can decide reachability of graphs of degree three [1].

One motivation for generalising the result of Cook & Rackoff was to construct graphs of small degree, in which the number of nodes a JAG can visit is not only smaller than the number of nodes in the graph, but also than the length of the shortest path between certain two nodes. The degree of these graphs should again depend only on the number

of pebbles. This rules out the graphs defined by Cook & Rackoff. We next construct a family of graphs $H_{m,d}$ with the desired property.

The undirected graph $H_{m,d}$ can be understood as arising from G_{m,m^d} by replacing each node with a copy of $G_{m,d}$. This is illustrated for $m = d = 2$ in the figure depicting $G_{2,2^2}$ and $H_{2,2}$ below. With this definition, $H_{m,d}$ has a large diameter like G_{m,m^d}, but unlike G_{m,m^d} the degree of $H_{m,d}$ exceeds that of $G_{m,d}$ only by two.



Formally, $H_{m,d}$ has node set $(\mathbb{Z}/m\mathbb{Z})^{V_{m,d}} \times V_{m,d}$, where $V_{m,d}$ is the node set of $G_{m,d}$. There are edges between $\langle f, x\rangle$ and $\langle f, x + g_i\rangle$ for all $i \in \{1, \ldots, d\}$ and also edges between $\langle f, x\rangle$ and $\langle(\lambda i.\,\text{if } i = x \text{ then } f(i) + 1 \text{ else } f(i))\ , x\rangle$. Notice how in the last edge, the second component of the pair acts as an address for the increment.

The graph $H_{m,d}$ may be described as the Cayley graph of a *wreath product*. We recall (a special case of) this concept here. Let A and B be groups. The wreath product $A \wr B$ is a group with carrier set $B^A \times A$, where B^A denotes the set of all functions from the carrier set of A to that of B. The multiplication is defined by

$$\langle f, x\rangle \cdot \langle g, y\rangle = \langle \lambda i.\, f(i) \cdot g(x^{-1} \cdot i),\ x \cdot y\rangle.$$

If $g_1^A, \ldots, g_n^A$ and $g_1^B, \ldots, g_m^B$ are generators of the groups A and B respectively, then $\{\langle \lambda i.\, e_B,\ g_i^A\rangle \mid 1 \le i \le n\} \cup \{\langle \lambda i.\,\text{if } i = e_A \text{ then } g_j^B \text{ else } e_B,\ e_A\rangle \mid 1 \le j \le m\}$ is a set of generators for $A \wr B$. Furthermore, if A has exponent m_A, B has exponent m_B and B is commutative then $A \wr B$ has exponent $m_A \cdot m_B$.

Example 4. For positive natural numbers m and d define the free action graph $H_{m,d}$ to be the Cayley graph on the group $D_{m,d} \wr (\mathbb{Z}/m\mathbb{Z})$ with respect to the elements $g_1, \ldots, g_{2(d+1)}$ defined as follows. For the elements $g_1, \ldots, g_{d+1}$ take generators of $D_{m,d} \wr (\mathbb{Z}/m\mathbb{Z})$, obtained from those of $D_{m,d}$ and $\mathbb{Z}/m\mathbb{Z}$ as described above. The rest of the elements are defined by $g_{i+(d+1)} = g_i^{-1}$ for all $i \in \{1, \ldots, d+1\}$.

The choice of $\vec{g}$ as a list of generators that is closed under inverses makes $H_{m,d}$ a connected undirected graph. Its exponent exponent is m^2. The diameter of $H_{m,d}$ is at least m^d, since for all nodes x and all $i \in \{1, \ldots, d+1\}$ the nodes x and $x \odot g_i$ are vectors that differ in at most one component, so that a path from $\langle \lambda i.\, 0,\ 0\rangle$ to $\langle \lambda i.\, 1,\ 1\rangle$ must have length at least m^d. □

Using Prop. 3 we now get that any JAG J with $|P|$ pebbles and $|Q|$ states can visit only $(m^2|Q|)^{c^{|P|}}$ nodes in $H_{m,d}$. By choosing $d > 3c^{|P|}$ and $m = |Q|$, we get that J can visit $(m^2|Q|)^{c^{|P|}} = m^{3c^{|P|}} < m^d$ nodes in $H_{m,d}$. But the diameter of $H_{m,d}$ is at least m^d, so that its diameter is greater than the number of nodes J can visit in it.

We note that one can further widen the gap between the nodes that J can visit and the diameter of the graph by iterating the construction of $H_{m,d}$ from $G_{m,d}$, that is by studying the Cayley graph for the wreath product $((D_{m,d} \wr (\mathbb{Z}/m\mathbb{Z})) \wr (\mathbb{Z}/m\mathbb{Z})$ etc.

4 Reachability

In this section we give a proof of Prop. 3, which we have formalised in Coq. We start by giving a proper definition of Jumping Automata on Graphs.

Definition 5 (Jumping Automaton on Graphs). *A* jumping automaton on graphs *is a triple* (Q, P, δ) *consisting of a finite set* Q *of states, a finite set* P *of pebbles and a function* $\delta\colon Q \times \Sigma_P \to Q \times M_P$*, where the set* Σ_P *of* observations *is the set of equivalence relations on* P *and* $M_P := (P \times P) \cup (P \times \mathbb{N})$ *is the set of* moves.

Thus a JAG is a finite state machine that in each step obtains an input of type Σ_P and makes an output of type M_P. The input in Σ_P is intended to contain the information which two pebbles lie on the same graph node. The intention for the moves is such that a move $\langle x, y\rangle \in P \times P$ represents the jump of pebble y to x and a move $\langle x, i\rangle \in P \times \mathbb{N}$ represents the move of pebble x along edge number i. We use the suggestive notation $[x{:=}y]$ and $[x{:=}succ_i(x)]$ for the two kinds of moves.

Concretely, a JAG can operate on a (Σ_P, M_P)-state space.

Definition 6 (State Space). *A* (Σ, M)-state space $(S, [-], \cdot)$ *consists of a* state set S*, an* observation function $[-]\colon S \to \Sigma$ *and a* move function $(-) \cdot (-)\colon S \times M \to S$.

We introduce the notion of state space not with the aim of generalising JAGs, but in order to make it clear which properties depend on the concrete construction of the state space. In the end, we are only interested in state spaces generated by graphs.

A locally ordered graph G with vertex set V gives rise to a (Σ_P, M_P)-state space $S_P(G)$ whose state set consists of all functions $\rho \in P \to V$ that place the pebbles on the graph nodes. The observation function $[-]$ maps a function ρ to the equivalence relation $[\rho] \in \Sigma_P$ defined by $\langle x, y\rangle \in [\rho] \iff \rho(x) = \rho(y)$. Thus, that a JAG can see whether or not any two pebbles lie on the same graph node. The move function $(-) \cdot (-)$ formalises pebble jumps and edge moves in the evident way.

A *configuration* of a JAG $J = (Q, P, \delta)$ on a state space $(S, [-], \cdot)$ is a pair $\langle q, \rho\rangle \in Q \times S$ of a machine state $q \in Q$ and an external state $\rho \in S$. Write $Conf_{J,S}$ for the set of configurations. The transition relation $\longrightarrow_{J,S}$ on configurations is the least relation such that $\delta(q, [\rho]) = \langle q', m\rangle$ implies $\langle q, \rho\rangle \longrightarrow_{J,S} \langle q', \rho \cdot m\rangle$. Write $\longrightarrow^*_{J,S}$ for the reflexive transitive closure of $\longrightarrow_{J,S}$. Since $\longrightarrow_{J,S}$ is a deterministic relation, we have, for any configuration C, a unique configuration that is reached from C in k steps. We write $C(k)$ for it.

4.1 Abstraction

We now work towards the proof of Prop. 3, which in its essence is that of Cook & Rackoff [1]. In this section, we define a framework for stating abstractly the properties that are needed in this proof.

As outlined in Sect. 2, the proof of Prop. 3 goes by giving a bound on the time when the moves of a JAG become periodic and showing that on a free action graph such a periodicity must soon lead to a configuration of the JAG being repeated. To give a bound on when the moves of a JAG become periodic, we formalise what a JAG could possibly learn about the positions of its pebbles on a free action graph. We capture this using the notion of abstraction that we introduce next.

Definition 7. *An* abstraction *of a (Σ, M)-space $(S, [-], \cdot)$ is a set X together with functions $init\colon S \to X$, $\omega_0\colon X \times M \to X$ and $\omega_1\colon X \times M \times \Sigma \to X$ and relations $\Vdash \subseteq X \times S$ and predictable $\subseteq X \times M \times \Sigma$, such that for all $\rho, \rho' \in S$, $\xi \in X$ and $m \in M$ the following properties all hold:*

1. $init(\rho) \Vdash \rho$.
2. $\xi \Vdash \rho$ *implies* $\omega_1(\xi, m, [\rho \cdot m]) \Vdash \rho \cdot m$.
3. $\xi \Vdash \rho$ *and* $\langle \xi, m, [\rho \cdot m]\rangle \in$ *predictable implies* $\omega_1(\xi, m, [\rho \cdot m]) = \omega_0(\xi, m)$.
4. $\xi \Vdash \rho$ *and* $\xi \Vdash \rho'$ *implies* $[\rho] = [\rho']$.

We think of the elements of X as the abstract knowledge that may be obtained about configurations. The statement $\xi \Vdash \rho$ expresses that the abstract knowledge ξ is consistent with configuration ρ. The value $init(\rho)$ represents the abstract knowledge that can be read off directly from ρ without any knowledge on the history of the computation.

There are two step functions ω_0 and ω_1 in order to distinguish between predictable and non-predictable steps. In a predictable step, the successor abstract state depends only on the previous abstract state and the move, while a non-predictable step may further depend on the observation in the state-space after the move. This reflects the distinction in Cook & Rackoff's proof between steps without a coalition and those with a coalition. When no coalition occurs then one can determine the known distances after the step from those before the step, while in the event of a coalition, one additionally needs to know which pebbles collided unexpectedly.

Extremal examples of abstractions can be obtained by letting $X = \Sigma$ and $X = S$, where in the former case no step is predictable, while in the latter all are.

For a JAG $J = (Q, P, \delta)$ and an abstraction X for state-space S, we define *extended configurations* to be configurations together with an abstract value of the state. The set of extended configurations is given by $Conf_{J,S}(X) = \{\langle q, \rho, \xi\rangle \in Q \times S \times X \mid \xi \Vdash \rho\}$.

The transition relation of the JAG J can now be partitioned into predictable and non-predictable steps, as given by the two rules below.

$$\text{(PREDICTABLE)}\ \frac{\delta(q, [\rho]) = \langle q', m\rangle \qquad \langle \xi, m, [\rho \cdot m]\rangle \in \mathit{predictable}}{\langle q,\ \rho,\ \xi\rangle \xrightarrow{=}_{J,S} \langle q',\ \rho \cdot m,\ \omega_0(\xi, m)\rangle}$$

$$\text{(NON-PREDICTABLE)}\ \frac{\delta(q, [\rho]) = \langle q', m\rangle \qquad \langle \xi, m, [\rho \cdot m]\rangle \notin \mathit{predictable}}{\langle q,\ \rho,\ \xi\rangle \xrightarrow{>}_{J,S} \langle q',\ \rho \cdot m,\ \omega_1(\xi, m, [\rho \cdot m])\rangle}$$

Clearly, both $E \xrightarrow{=} F$ and $E \xrightarrow{>} F$ imply $\pi_C E \longrightarrow \pi_C F$, where we write $\pi_C E$ for the projection from an extended configuration to the ordinary configuration it contains. We also write $\pi_Q E \in Q$, $\pi_S E \in S$ and $\pi_X E \in X$ for the evident projections.

In the rest of this paper we use the equivalence relations $\simeq_{QX}$ and $\simeq_{Q\Sigma}$ on extended configurations defined by:

$$E \simeq_{QX} F \iff \pi_Q E = \pi_Q F \ \wedge\ \pi_X E = \pi_X F$$
$$E \simeq_{Q\Sigma} F \iff \pi_Q E = \pi_Q F \ \wedge\ [\pi_S E] = [\pi_S F]$$

Note that we have $\simeq_{Q\Sigma} \subseteq \simeq_{QX}$ by item 4 in Def. 7. Furthermore, if $E \simeq_{Q\Sigma} F$ holds then the machine J will make the same moves from these two configurations, i.e. $\delta(\pi_Q E, [\pi_S E]) = \delta(\pi_Q F, [\pi_S F])$.

Lemma 8 (Determinacy of abstract values)

1. *If* $E \simeq_{QX} F$, $E \xrightarrow{=} E'$ *and* $F \xrightarrow{=} F'$ *hold, then so does* $E' \simeq_{QX} F'$.
2. *If* $E \simeq_{QX} F$, $E \xrightarrow{>} E'$, $F \xrightarrow{>} F'$ *and* $[\pi_S E'] = [\pi_S F']$, *then also* $E' \simeq_{QX} F'$.

4.2 Reachability

In this section we prove Prop. 3. Fix a free action graph $G = G(V, D, g_1, \ldots, g_n, \odot)$ with exponent m and a JAG $J = (Q, P, \delta)$. We omit subscripts J, P, G where possible.

First we define the instance of an abstraction that formalises the 'maximum possible knowledge' a JAG can obtain about a free action graph, as outlined above. The following lemma contains all the properties that we need of it.

Lemma 9. *There exists an abstraction* X *of the state-space* $S(G)$ *such that:*

1. *The function* $init\colon S(G) \to X$ *can be written as the composition of two functions* $init_1\colon S(G) \to \Sigma$ *and* $init_2\colon \Sigma \to X$.
2. *There exists a function* $[-]\colon X \to Eq(P)$, *such that if* $\xi \Vdash \rho$, $\xi' \Vdash \rho$ *and* $[\xi] = [\xi']$ *all hold then so does* $\xi = \xi'$.
3. *There exists a measure function* $\mu\colon Conf(X) \to \{1, \ldots, |P|\}$ *satisfying the following three properties for all* $E, F \in Conf(X)$, $C \in Conf$, $k \in \mathbb{N}$ *and* $\xi \in X$.
   (a) $E \xrightarrow{=}_{S(G)} F$ *implies* $\mu(F) = \mu(E)$.
   (b) $E \xrightarrow{>}_{S(G)} F$ *implies* $\mu(F) = \mu(E) - 1$.
   (c) $\mu\left(\langle C, init(\pi_S C)\rangle(k)\right) \geq \mu\left(\langle C, \xi\rangle(k)\right)$.

Proof. The abstraction X is the set of partial functions $P \times P \rightharpoonup D$ whose domain is an equivalence relation. The functions describe partial knowledge about the relative displacement of any two pebbles. This is formalised by the relation $\Vdash$:

$$\begin{aligned} \xi \Vdash \rho \quad \iff \quad & \forall x, y \in P.\, (\xi(x,y) \neq \bot \implies \rho(y) = \rho(x) \odot \xi(x,y)) \\ & \wedge\, (\rho(x) = \rho(y) \in V \implies \xi(x,y) = e_D) \end{aligned}$$

The functions ω_0 and ω_1 return the updated knowledge of the relative distances after a move. The two functions ω_0 and ω_1 differ only in their handling of edge moves $[x{:=}succ_i(x)]$. In an edge move, it is possible that two pebbles collide whose relative distance is not yet known. In the function ω_0 it is assumed that such a collision does not happen, while ω_1 handles this possibility. Note that in order to tell whether a collision has occurred, it is necessary to observe the state-space after the move. Hence, collisions can only be accounted for in ω_1.

The function ω_0 is defined by the equations below, which hold for all $\xi \in X$, $x, y \in P$, $u, v \in P \setminus \{x\}$ and $i \in \mathbb{N}$, and in which we let $g_i := e_D$ if i exceeds the degree of G.

$$\begin{aligned} \omega_0\left(\xi, [x{:=}y]\right)(x,x) &= e_D & \omega_0\left(\xi, [x{:=}succ_i(x)]\right)(x,x) &= e_D \\ \omega_0\left(\xi, [x{:=}y]\right)(x,v) &= \xi(y,v) & \omega_0\left(\xi, [x{:=}succ_i(x)]\right)(x,v) &= g_i^{-1} \cdot \xi(x,v) \\ \omega_0\left(\xi, [x{:=}y]\right)(u,x) &= \xi(u,y) & \omega_0\left(\xi, [x{:=}succ_i(x)]\right)(u,x) &= \xi(u,x) \cdot g_i \\ \omega_0\left(\xi, [x{:=}y]\right)(u,v) &= \xi(u,v) & \omega_0\left(\xi, [x{:=}succ_i(x)]\right)(u,v) &= \xi(u,v) \end{aligned}$$

Here, we follow the convention that any expression containing an undefined subexpression is itself undefined. We omit the definition of ω_1, which extends that of ω_0 by a special case to update the known relative distances in case of an unpredicted collision.

The function $init$ is defined such that $init(\rho)(x,y) = e_D$ holds when $x[\rho]y$ does and $init(\rho)(x,y)$ is undefined for all other $x, y \in P$. For $\mu(q,\rho,\xi)$ we take the number of equivalence classes in the domain of ξ. Finally, we define the set $predictable$ to contain those triples $\langle \xi, m, \sigma\rangle$ for which the measure of ξ is the same as that of $\omega_1(\xi, m, \sigma)$.

The required properties then follow by a straightforward (but tedious and lengthy) calculation from the properties of free action graphs. □

Having proved this lemma, it remains to observe that the proof of Cook & Rackoff can be carried out with just the abstract structure defined in the lemma.

Let $k \in \mathbb{N}$. In this section we first establish a bound on the number of configurations that appear in computations of up to $k+1$ steps. Since k is arbitrary and the bound will not depend on k, this shall be enough to bound the number of configurations in computation sequences of arbitrary length.

The proof of the bound goes by induction on how often in a computation sequence the measure from Lemma 9 strictly decreases. The points of decrease are called *coalitions*, in reference to the fact that in the abstract values from the proof of Lemma 9 two equivalence classes are united at these points.

Definition 10 (Coalition). *Let $E \in Conf(X)$ and $n \in \mathbb{N}$. The n-*th coalition *is the smallest number $coal_n E \leq k+1$ satisfying $\mu(E(coal_n E)) \leq P - n$ or $coal_n E = k+1$.*

We aim to give an upper bound on the number A_n defined by

$$A_n := \max_{C \in Conf} |\{C(i) \mid i < coal_{n+1}\langle C, init(C)\rangle\}|\,.$$

To do so, we also need to give an upper bound on the number of different possible behaviours of J. Similarity of behaviour up to the n-th coalition is captured by the equivalence relation $\sim_n \subseteq Conf(X) \times Conf(X)$ defined by

$$E \sim_n F \iff \forall i \leq n.\ coal_i E = coal_i F \ \wedge\ E(coal_i E) \simeq_{QX} F(coal_i F).$$

If E and F are $\sim_n$-related then the JAG J makes the same moves in the computation sequences starting with E and F, up to any time before the $(n+1)$-th coalition in either sequence. This follows from the next lemma, which is a consequence of Lemma 8 and the fact that J always makes the same moves from $\simeq_{Q\Sigma}$-related configurations.

Lemma 11. *If $E \sim_n F$ and $i < coal_{n+1}E$ and $i < coal_{n+1}F$ all hold then so does $E(i) \simeq_{QX} F(i)$ and $E(i) \simeq_{Q\Sigma} F(i)$.*

The following necessary condition for $E \sim_{n+1} F$ also follows using Lemma 8.

Lemma 12. *To show $E \sim_{n+1} F$ it suffices to show that $E \sim_n F$, $coal_{n+1}E = coal_{n+1}F$ and $[\pi_S E(coal_{n+1}E)] = [\pi_S F(coal_{n+1}F)]$ all hold.*

Define an equivalence relation $\approx_n$ for similar behaviour on ordinary configurations:

$$C \approx_n C' \iff \langle C, init(\pi_S C)\rangle \sim_n \langle C', init(\pi_S C')\rangle.$$

Let R_n be the number of equivalence classes of $\approx_n$.

We can now bound the numbers R_n and A_n by induction on n. We start with the bound on R_n. The bound on A_n appears in Lemma 16 below.

Lemma 13. *For all $n \in \mathbb{N}$, the following two inequalities hold.*

$$R_0 \leq |Q| \cdot |P|^{|P|} \tag{1}$$

$$R_{n+1} \leq R_n \cdot (2 + A_n \cdot |P|^{|P|}) \cdot |P|^{|P|} \tag{2}$$

Proof. Ad (1). We have $coal_0(E) = 0$ for all E. We therefore know that $C \approx_0 C'$ is equivalent to $\pi_Q C = \pi_Q C'$ and $init(\pi_S C) = init(\pi_S C')$. The result follows since there are only $|P|^{|P|}$ possibilities for $init(\pi_S C)$, which follows from Lemma 9.1 and because there are at most $|P|^{|P|}$ equivalence classes on P.

Ad (2). We use Lemma 12 to decompose the equivalence classes of $\approx_{n+1}$. The asserted bound then follows because: (i) The equivalence relation $\approx_n$ has at most R_n equivalence classes. (ii) There are $(2 + A_n \cdot |P|^{|P|})$ possibilities for $coal_{n+1}(E)$. It can be either $k+1$ or must be below $1 + A_n \cdot |P|^{|P|}$, since after at most $1 + A_n \cdot |P|^{|P|}$ steps a configuration will be repeated, by definition of A_n and Lemma 9.2, and if a repeat appears then the $(n+1)$-th coalition must be $k+1$. (iii) Finally, $[\pi_S E(coal_{n+1}E)]$ has at most $|P|^{|P|}$ possible values. □

For any list of moves $\alpha \in M^*$, an action on pebble assignments $\rho \in S(G)$ can be defined by $\rho \cdot \varepsilon = \rho$ and $\rho \cdot (m\alpha) = (\rho \cdot m) \cdot \alpha$. With this notation we have the following lemma, whose proof is just like that of Lemma 4.6 of [1].

Lemma 14. *Let G be a free action graph with exponent m. Let $\alpha \in M^*$ be a sequence of moves. Then the sequence $(\rho_i \in S(G))_{i \geq 0}$ defined by $\rho_0 = \rho$ and $\rho_{i+1} = \rho_i \cdot \alpha$ has the property $\rho_{|P|} = \rho_{|P|+m \cdot |P|!}$.*

Lemma 15. *If $\pi_C E \approx_n \pi_C F$ and $i < coal_{n+1}E$ and $i < coal_{n+1}F$ all hold then so does $E(i) \simeq_{Q\Sigma} F(i)$.*

Proof. Using Lemma 9.3.(c) one gets $i < coal_{n+1}E \leq coal_{n+1}\langle \pi_C E, init(\pi_S E)\rangle$ and likewise for F. This implies $\langle \pi_C E, init(\pi_S E)\rangle(i) \simeq_{Q\Sigma} \langle \pi_C F, init(\pi_S F)\rangle(i)$ by Lemma 11. But the assertion follows from this, because it is easily seen that $\pi_C E(i) = \pi_C(\langle \pi_C E, init(\pi_S E)\rangle(i))$ and a similar equation for F hold and that $\pi_C E(i) = \pi_C F(i)$ implies $E(i) \simeq_{Q\Sigma} F(i)$.

Lemma 16. *For all n the inequality $A_n \leq (1 + |P| + m \cdot |P|!) \cdot R_n$ holds.*

Proof. Consider a computation $E \longrightarrow E(1) \longrightarrow \ldots \longrightarrow E(l)$ with $l < coal_n E$. Suppose, for a contradiction, there are more than $(1 + |P| + m \cdot |P|!) \cdot R_n$ configurations in $\{\pi_C E(0), \ldots, \pi_C E(l)\}$. Then, by definition of R_n, there exist i and j with $1 \leq i < j \leq R_n$ and $\pi_C E(i) \approx_n \pi_C E(j)$. Using Lemma 15, this implies $E(r) \simeq_{Q\Sigma} E(r + (j - i))$ for all r with $i \leq r \leq l - (j - i)$. Hence, there is a sequence of moves α of length $(j - i) \leq R_n$ that is repeated from point $E(i)$ onwards. By Lemma 14, it follows that some configuration appears twice in the list $\pi_C E(i), \pi_C E(i + (j - i)), \ldots, \pi_C E(i + (|P| + m \cdot |P|!) \cdot (j - i))$. The JAG must therefore go into a loop after at most $(1 + |P| + m \cdot |P|!) \cdot R_n$ steps, which implies the required contradiction. □

Proof (of Prop. 3). Using Lemmas 13 and 16, it is straightforward to find a constant c, such that $A_{|P|} \leq (Qm)^{c^{|P|}}$ holds. (In Coq we have $c = 2^{32}$ by a crude estimation.) The required assertion follows from this, since c does not depend on k and we have $coal_{|P|+1}E = k + 1$ by the properties of μ. □

Up to this point, the results in this section have been fully formalised in Coq, where we have excluded uninteresting trivial cases by assuming $0 < |Q|$, $1 < m$ and $1 < |P|$.

Corollary 17 (Cook & Rackoff 1980). *For each finite set P there is a d such that no* JAG $J = (Q, P, \delta)$ *decides s-t-reachability on undirected graphs of degree d.*

Proof. Construct a graph of two disjoint copies of $H_{m,d}$ and connect the two nodes given by vectors with $(m-1)$ in all components. Let s and t be the two nodes consisting of all zeros. If m and d are large enough then J, when started with each pebble on one of the two nodes s or t, cannot reach the edge connecting the two copies of $H_{m,d}$, since its distance from s and t in $H_{m,d}$ is larger than the number of nodes that J can visit. Hence, J cannot distinguish this graph from the one with this edge removed. □

Corollary 18 (Cook & Rackoff 1980). *No* JAG *decides s-t-reachability on undirected graphs of degree* 3.

5 Formalisation in Coq

One of the main contributions of this paper is the formalisation of the results in the previous section, up to and including the proof of Prop. 3 (as remarked above, we elude trivial cases in the formalisation by assuming $0 < |Q|$, $1 < m$ and $1 < |P|$). The formalisation also comprises the construction of the graphs $G_{m,d}$ and $H_{m,d}$ from Sect. 3.

The presentation in the previous section can be understood as an overview of the formalisation[1] in ordinary mathematical notation. Indeed, most of the lemmas in the previous section correspond directly to lemmas in the Coq development. To give a concrete example, Lemma 12 appears in the formalisation in the following ASCII notation.

```
Lemma decomp_sim: forall k n E1 E2,
 (sim k n E1 E2) -> (coal k n.+1 E1 == coal k n.+1 E2)
  -> observe (piS (coalE k n.+1 E1))
      = observe (piS (coalE k n.+1 E2))
  -> (sim k n.+1 E1 E2).
```

The proofs in the formalisation also follow closely the informal outline in the previous section. The main difficulty in making it possible for the formal development to remain close to the informal proofs has been to define the basic types in an appropriate way. In the last section we have used classical logic, which appears to be at odds with Coq's intuitionistic meta-logic, and we have used counting arguments, such as that there are at most $|P|^{|P|}$ equivalence relations on a finite set P, which are not readily available in Coq. In this section we discuss the basic choices that have allowed us to faithfully formalise the proofs from the last section.

[1] Available from: `http://www.tcs.ifi.lmu.de/~schoepp/formalcr.html`

The formalisation consists of approximately 5000 lines of code. It is written in the tactic language SSReflect 1.1 developed by Gonthier, Mahboubi and Théry [7,6], which has its origin in Gonthier's formalisation of the four-colour theorem [4]. We have chosen this tactic language for its conciseness and expressivity and because it contains an excellent library for working with finite sets.

Small-scale Reflection. Based on his experience from formalising the four colour theorem [4], Georges Gonthier has advocated a proof methodology that emphasises the use of computation at the propositional level [5]. One central idea is to express the truth of a predicate φ on some structured data type A that one wants to work with as a computational problem. This means that one writes a function $f\colon A \to \texttt{bool}$, where `bool` is the data type with elements `true` and `false`, such that $f(x)$ is `true` if and only if the logical proposition $\varphi(x)$ holds. The function f may implement a decision procedure, for example. Then, one proves correctness of the decision procedure in the logic, i.e. that $\varphi(x)$ holds if and only if $f(x) = \texttt{true}$ does. As a result, if one wants to prove $\varphi(M)$ for some particular element M, one may replace this goal by $f(M) = \texttt{true}$ and have the proof system reduce the function f. If M is a closed value having the property φ then $f(M)$ will reduce to `true`, leaving the trivial goal $\texttt{true} = \texttt{true}$. Even if M is not a closed value, it is in many cases still possible to reduce $f(M)$ partially and so make progress in a proof. In this way, propositions are proved by evaluating functions.

The general idea then is to write predicates as functional programs and to prove their correctness. Since boolean-valued functions are very often used in place of predicates, there is a coercion so that one can just write $f(x)$ instead of $f(x) = \texttt{true}$. The tactic language SSReflect provides a convenient environment for carrying out proofs with such boolean-valued predicates in Coq. For instance, in our formalisation we have used a function $\texttt{iforall}\colon \forall A\colon \texttt{finType}.\ (A \to \texttt{bool}) \to \texttt{bool}$ of universal quantification on finite sets. SSReflect provides the convenient concept of views that, after proving the equivalence of $\texttt{iforall}(f)$ and $\forall x.\ f(x)$ once, allows one to treat occurrences of $\texttt{iforall}(f)$ in essentially the same way as an ordinary quantification.

The approach of modelling predicates as `bool`-valued functions has been very important for the formalisation in this paper. Since `bool`-valued functions must return either `true` or `false`, they validate the law of excluded middle, and we have used this fact many times to formalise the classical arguments in this paper. Also, our construction of quotients on finite types, described below, relies on predicates being `bool`-valued.

SSReflect turned out to be very well-suited for working with finite sets and decidable predicates in a way that is similar to classical logic. It contains a library of finite sets, which are being modelled as types with decidable equality and an enumeration of their elements as a (necessarily finite) list. With this library it is possible to prove, for example, the following pigeonhole principle in a few lines and in a way that is very close to a standard informal classical proof.

$$\texttt{Lemma pigeon}\colon\ \forall d_1\ d_2\colon \texttt{finType},\ f\colon d_1 \to d_2.$$
$$(\texttt{card}\ d_2 < \texttt{card}\ d_1) \to (\exists x, y\colon d_1.\ (\texttt{negb}\ (x == y))\ \&\&\ (f\ x == f\ y))$$

Here, $==$ denotes decidable equality, `negb` stands for negation and $\&\&$ for conjunction.

This example illustrates that by using boolean-valued predicates, we can formalise classical arguments as conveniently as in a theorem prover with classical logic, such as,

say, Isabelle/HOL, while at the same time being able to use Coq's convenient dependently typed programming facilities.

Finite Functions. The formalisation of the proofs from Sect. 4 depends, among other things, on being able to count the number of configurations that appear in a computation sequence. Since configurations are modelled by functions from pebbles to graph nodes, we must therefore be able to count functions with finite domain and codomain.

To do this, we use the intensional representation of finite functions by their graphs proposed by Gonthier et al. [8]. However, we use functions with finite codomain as opposed to just a decidable codomain in [8], since we also need to prove cardinality properties, such as that the finite function space $X \to Y$ has cardinality $|Y|^{|X|}$.

An intensional representation of functions is useful also to avoid problems with Coq's equality being intensional, as observed in [8]. Two finite functions $f, g\colon X \to Y$ are extensionally equal if and only if their intensional representations are equal in Coq's intensional equality. Since one can write in Coq functions to convert a finite function to its intensional representation and vice versa, one can therefore work with the intensional representation of functions instead of functions themselves. With implicit coercions, one can treat intensionally represented functions almost like ordinary ones.

It is furthermore not hard to show that the functions between finite sets form themselves a finite set with decidable equality. As a result, finite functions may be used in the construction of finite sets, which is useful for defining finite wreath products, for example. Another example, where decidable equality $==$ on finite functions is useful, is the definition of the function `iforall`, which decides whether a given boolean-valued predicate holds for all elements of a finite set. It may be defined simply by

$$\begin{aligned}&\texttt{iforall}(A\colon \texttt{finType})(f\colon A \to \texttt{bool})\colon \texttt{bool} :=\\ &\qquad (\texttt{fgraph\_of\_fun}\ f) == (\texttt{fgraph\_of\_fun}\ (\lambda x.\,\texttt{true})),\end{aligned}$$

where `fgraph_of_fun` converts a function to its intensional representation.

Equivalence Relations and Quotients. The proofs in Sect. 4 rely on being able to count the equivalence classes of $\approx_n$ and to show that there are at most $|P|^{|P|}$ equivalence relations on finite P. To formalise such arguments, we have developed a representation of equivalence relations and quotients on finite sets.

We represent equivalence relations on a finite set A by functions of type $A \to A \to \texttt{bool}$ together with proofs of reflexivity, symmetry and transitivity. Using a boolean-valued function to represent relations allows us to give an intensional representation of equivalence relations, just like for finite functions, and to define quotient types.

To define an intensional representation of equivalence relations, we use a choice function for finite sets provided by SSReflect. This choice function computes, for each finite set A and each predicate $f\colon A \to \texttt{bool}$, an element a of A that satisfies $f(a)$, if such an element exists. In particular, suppose R of type $A \to A \to \texttt{bool}$ is an equivalence relation. The equivalence class of a is given by $R(a)\colon A \to \texttt{bool}$. Using the choice function, we can pick a canonical element from this equivalence class. In this way, we can represent the equivalence relation R by a finite function of type $A \to A$, which maps each element of A to a canonical representative of its equivalence class.

For the intensional representation of equivalence classes, we then simply take the choice functions $A \to A$ that arise from this construction. In particular, using the results

on finite functions, we can define it as a finite data type with decidable equality. It follows immediately that there can be no more than $|A|^{|A|}$ equivalence relations on A.

With this groundwork, it is easy to construct quotient types. We define the quotient A/R simply as the image of the choice function $A \to A$ that represents R. This being a finite type, we can use it to count the number of equivalence classes, as is required to formalise the proof of Cook & Rackoff. All in all, we can work with finite quotients just as we are used to in informal work.

The construction of the intensional representation of equivalence relations relies on relations being modelled as boolean-valued functions $A \to A \to \texttt{bool}$. It would not work with `Prop` instead of `bool`, since no choice function would be available then. The relations from Def. 7 are therefore all modelled as boolean-valued functions.

6 Conclusion

Once the right basic definitions had been found, the formalisation of the main results using Coq and SSReflect has been surprisingly smooth. We hope that this positive experience can be repeated in the formalisation of further results from complexity theory. We are aware only of little existing work in this direction, e.g. [3]. By formalising a non-trivial result from complexity theory we have given evidence that the current theorem prover technology is ready for this task.

Acknowledgements. I wish to thank Martin Hofmann for interesting discussions and anonymous referees for their constructive comments. This work was supported by the DFG project Pro.Platz (programming language aspects of sublinear space complexity).

References

1. Cook, S.A., Rackoff, C.: Space lower bounds for maze threadability on restricted machines. SIAM Journal of Computing 9(3), 636–652 (1980)
2. Etessami, K., Immerman, N.: Reachability and the power of local ordering. Theoretical Computer Science 148(2), 261–279 (1995)
3. Gamboa, R., Cowles, J.R.: A mechanical proof of the Cook-Levin theorem. In: Slind, K., Bunker, A., Gopalakrishnan, G.C. (eds.) TPHOLs 2004, vol. 3223, pp. 99–116. Springer, Heidelberg (2004)
4. Gonthier, G.: A computer-checked proof of the four-colour theorem. Technical report, `http://research.microsoft.com/~gonthier/4colproof.pdf`
5. Gonthier, G.: Notations of the four colour theorem proof. Technical report, `http://research.microsoft.com/~gonthier/4colnotations.pdf`
6. Gonthier, G., Mahboubi, A.: A small scale reflection extension for the Coq system. INRIA Technical Report (December 2007)
7. Gonthier, G., Mahboubi, A., Théry, L.: SSReflect extension for Coq, Version 1.1, `http://www.msr-inria.inria.fr/Projects/math-components`
8. Gonthier, G., Mahboubi, A., Rideau, L., Tassi, E., Théry, L.: A modular formalisation of finite group theory. In: TPHOLs, pp. 86–101 (2007)
9. Lu, P., Zhang, J., Poon, C.K., Cai, J.: Simulating undirected st-connectivity algorithms on uniform JAGs and NNJAGs. In: ISAAC, pp. 767–776 (2005)
10. Reingold, O.: Undirected st-connectivity in log-space. In: STOC, pp. 376–385 (2005)

Improving Context-Sensitive Dependency Pairs*

Beatriz Alarcón[1], Fabian Emmes[2], Carsten Fuhs[2], Jürgen Giesl[2],
Raúl Gutiérrez[1], Salvador Lucas[1],
Peter Schneider-Kamp[2], and René Thiemann[3]

[1] DSIC, Universidad Politécnica de Valencia, Spain
[2] LuFG Informatik 2, RWTH Aachen University, Germany
[3] Institute of Computer Science, University of Innsbruck, Austria

Abstract. Context-sensitive dependency pairs (CS-DPs) are currently the most powerful method for automated termination analysis of context-sensitive rewriting. However, compared to DPs for ordinary rewriting, CS-DPs suffer from two main drawbacks: (a) CS-DPs can be *collapsing*. This complicates the handling of CS-DPs and makes them less powerful in practice. (b) There does not exist a "*DP framework*" for CS-DPs which would allow one to apply them in a flexible and modular way. This paper solves drawback (a) by introducing a new definition of CS-DPs. With our definition, CS-DPs are always non-collapsing and thus, they can be handled like ordinary DPs. This allows us to solve drawback (b) as well, i.e., we extend the existing DP framework for ordinary DPs to context-sensitive rewriting. We implemented our results in the tool AProVE and successfully evaluated them on a large collection of examples.

1 Introduction

Context-sensitive rewriting [23,24] models evaluations in programming languages. It uses a *replacement map* μ with $\mu(f) \subseteq \{1, ..., \mathrm{arity}(f)\}$ for every function symbol f to specify the argument positions of f where rewriting may take place.

Example 1. Consider this context-sensitive term rewrite system (CS-TRS)

$$\begin{array}{rlrl}
\mathsf{gt}(0,y) &\to \mathsf{false} & \mathsf{p}(0) &\to 0\\
\mathsf{gt}(\mathsf{s}(x),0) &\to \mathsf{true} & \mathsf{p}(\mathsf{s}(x)) &\to x\\
\mathsf{gt}(\mathsf{s}(x),\mathsf{s}(y)) &\to \mathsf{gt}(x,y) & \mathsf{minus}(x,y) &\to \mathsf{if}(\mathsf{gt}(y,0),\mathsf{minus}(\mathsf{p}(x),\mathsf{p}(y)),x)\\
\mathsf{if}(\mathsf{true},x,y) &\to x & \mathsf{div}(0,\mathsf{s}(y)) &\to 0\\
\mathsf{if}(\mathsf{false},x,y) &\to y & \mathsf{div}(\mathsf{s}(x),\mathsf{s}(y)) &\to \mathsf{s}(\mathsf{div}(\mathsf{minus}(x,y),\mathsf{s}(y)))
\end{array} \quad (1)$$

with $\mu(\mathsf{if}) = \{1\}$ and $\mu(f) = \{1, \ldots, \mathrm{arity}(f)\}$ for all other symbols f to model the usual behavior of if*: in $\mathsf{if}(t_1, t_2, t_3)$, one may evaluate t_1, but not t_2 or t_3. It will turn out that due to μ, this CS-TRS is indeed terminating. In contrast, if one allows arbitrary reductions, then the TRS would benon-terminating:*

* Authors from Valencia were partially supported by the EU (FEDER) and the Spanish MEC/MICINN, under grants TIN 2007-68093-C02-02 and HA 2006-0007. B. Alarcón was partially supported by the Spanish MEC/MICINN under FPU grant AP2005-3399. R. Gutiérrez was partially supported by the Spanish MEC/MICINN, under grant TIN 2004-7943-C04-02. Authors from Aachen were supported by the DAAD under grant D/06/12785 and by the DFG under grant GI 274/5-2.

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 636–651, 2008.

$$\mathsf{minus}(0,0) \to^+ \mathsf{if}(\mathsf{gt}(0,0),\mathsf{minus}(0,0),0) \to^+ \mathsf{if}(\ldots,\mathsf{if}(\mathsf{gt}(0,0),\mathsf{minus}(0,0),0),\ldots) \to^+ \ldots$$

There are two approaches to prove termination of context-sensitive rewriting. The first approach transforms CS-TRSs to ordinary TRSs, cf. [13,26]. But transformations often generate complicated TRSs where all termination tools fail.

Therefore, it is more promising to adapt existing termination techniques from ordinary term rewriting to the context-sensitive setting. Such adaptions were done for classical methods like RPO or polynomial orders [8,19,25]. However, much more powerful techniques like the *dependency pair* (DP) method [6] are implemented in almost all current termination tools for TRSs. But for a long time, it was not clear how to adapt the DP method to context-sensitive rewriting.

This was solved first in [1]. The corresponding implementation in the tool MU-TERM [3] outperformed all previous tools for termination of CS rewriting.

Nevertheless, the existing results on CS-DPs [1,2,4,20] still have major disadvantages compared to the DP method for ordinary rewriting, since CS-DPs can be *collapsing*. To handle such DPs, one has to impose strong requirements which make the CS-DP method quite weak and which make it difficult to extend refined termination techniques based on DPs to the CS case. In particular, the *DP framework* [14,17,21], which is the most powerful formulation of the DP method for ordinary TRSs, has not yet been adapted to the CS setting.

In this paper, we solve these problems. After presenting preliminaries in Sect. 2, we introduce a new notion of *non-collapsing* CS-DPs in Sect. 3. This new notion makes it much easier to adapt termination techniques based on DPs to context-sensitive rewriting. Therefore, Sect. 4 extends the *DP framework* to the context-sensitive setting and shows that existing methods from this framework only need minor changes to apply them to context-sensitive rewriting.

All our results are implemented in the termination prover AProVE [16]. As shown by the empirical evaluation in Sect. 5, our contributions improve the power of automated termination analysis for context-sensitive rewriting substantially.

2 Context-Sensitive Rewriting and CS-Dependency Pairs

See [7] and [23] for basics on term rewriting and context-sensitive rewriting, respectively. Let $\mathcal{P}os(s)$ be the set of *positions* of a term s. For a replacement map μ, we define the *active positions* $\mathcal{P}os^\mu(s)$: For $x \in \mathcal{V}$ let $\mathcal{P}os^\mu(x) = \{\varepsilon\}$ where ε is the root position. Moreover, $\mathcal{P}os^\mu(f(s_1,\ldots,s_n)) = \{\varepsilon\} \cup \{i\,p \mid i \in \mu(f),\ p \in \mathcal{P}os^\mu(s_i)\}$. We say that $s \trianglerighteq_\mu t$ holds if $t = s|_p$ for some $p \in \mathcal{P}os^\mu(s)$ and $s \triangleright_\mu t$ if $s \trianglerighteq_\mu t$ and $s \neq t$. Moreover, $s \triangleright_\mu t$ if $t = s|_p$ for some $p \in \mathcal{P}os(s) \setminus \mathcal{P}os^\mu(s)$. We denote the ordinary subterm relations by $\trianglerighteq$ and $\triangleright$.

A *CS-TRS* $(\mathcal{R},\mu)$ consists of a finite TRS $\mathcal{R}$ and a replacement map μ. We have $s \hookrightarrow_{\mathcal{R},\mu} t$ iff there are $\ell \to r \in \mathcal{R}$, $p \in \mathcal{P}os^\mu(s)$, and a substitution σ with $s|_p = \sigma(\ell)$ and $t = s[\sigma(r)]_p$. This reduction is an *innermost* step (denoted $\overset{i}{\hookrightarrow}_{\mathcal{R},\mu}$) if all t with $s|_p \triangleright_\mu t$ are in normal form w.r.t. $(\mathcal{R},\mu)$. A term s is in *normal form* w.r.t. $(\mathcal{R},\mu)$ if there is no term t with $s \hookrightarrow_{\mathcal{R},\mu} t$. A CS-TRS $(\mathcal{R},\mu)$ is *terminating* if $\hookrightarrow_{\mathcal{R},\mu}$ is well founded and *innermost terminating* if $\overset{i}{\hookrightarrow}_{\mathcal{R},\mu}$ is well founded.

Let $\mathcal{D} = \{root(\ell) \mid \ell \to r \in \mathcal{R}\}$ be the set of *defined symbols*. For every $f \in \mathcal{D}$, let $f^\sharp$ be a fresh *tuple symbol* of same arity, where we often write "F" instead of "$f^\sharp$". For $t = f(t_1, \ldots, t_n)$ with $f \in \mathcal{D}$, let $t^\sharp = f^\sharp(t_1, \ldots, t_n)$.

Definition 2 (CS-DPs [1]). *Let $(\mathcal{R}, \mu)$ be a CS-TRS. If $\ell \to r \in \mathcal{R}$, $r \trianglerighteq_\mu t$, and* $\mathrm{root}(t) \in \mathcal{D}$, *then $\ell^\sharp \to t^\sharp$ is an* ordinary dependency pair.[1] *If $\ell \to r \in \mathcal{R}$, $r \trianglerighteq_\mu x$ for a variable x, and $\ell \not\rhd_\mu x$, then $\ell^\sharp \to x$ is a* collapsing DP. *Let $\mathsf{DP}_o(\mathcal{R}, \mu)$ and $\mathsf{DP}_c(\mathcal{R}, \mu)$ be the sets of all ordinary resp. all collapsing DPs.*

Example 3. For the TRS of Ex. 1, we obtain the following CS-DPs.

$$\mathsf{GT}(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{GT}(x, y) \quad (2)$$
$$\mathsf{IF}(\mathsf{true}, x, y) \to x \quad (3)$$
$$\mathsf{IF}(\mathsf{false}, x, y) \to y \quad (4)$$
$$\mathsf{M}(x, y) \to \mathsf{IF}(\mathsf{gt}(y, \mathsf{0}), \mathsf{minus}(\mathsf{p}(x), \mathsf{p}(y)), x) \quad (5)$$
$$\mathsf{M}(x, y) \to \mathsf{GT}(y, \mathsf{0}) \quad (6)$$
$$\mathsf{D}(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{D}(\mathsf{minus}(x, y), \mathsf{s}(y)) \quad (7)$$
$$\mathsf{D}(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{M}(x, y) \quad (8)$$

To prove termination, one has to show that there is no infinite *chain* of DPs. For ordinary rewriting, a sequence $s_1 \to t_1, s_2 \to t_2, \ldots$ of DPs is a *chain* if there is a substitution σ such that $t_i\sigma$ reduces to $s_{i+1}\sigma$.[2] If all $t_i\sigma$ are terminating, then the chain is *minimal* [14,17,22]. But due to the collapsing DPs, the notion of "chains" has to be adapted when it is used with CS-DPs [1]. If $s_i \to t_i$ is a collapsing DP (i.e., if $t_i \in \mathcal{V}$), then instead of $t_i\sigma \hookrightarrow^*_{\mathcal{R},\mu} s_{i+1}\sigma$ (and termination of $t_i\sigma$ for minimality), one requires that there is a term w_i with $t_i\sigma \trianglerighteq_\mu w_i$ and $w_i^\sharp \hookrightarrow^*_{\mathcal{R},\mu} s_{i+1}\sigma$. For minimal chains, $w_i^\sharp$ must be terminating.

Example 4. Ex. 1 has the chain (5), (3), (5) as $\mathsf{IF}(\mathsf{gt}(\mathsf{s}(y), \mathsf{0}), \mathsf{minus}(\mathsf{p}(x), \mathsf{p}(\mathsf{s}(y))), x) \hookrightarrow^*_{\mathcal{R},\mu} \mathsf{IF}(\mathsf{true}, \mathsf{minus}(\mathsf{p}(x), \mathsf{p}(\mathsf{s}(y))), x) \hookrightarrow_{(3),\mu} \mathsf{minus}(\mathsf{p}(x), \mathsf{p}(\mathsf{s}(y)))$ *and* $(\mathsf{minus}(\mathsf{p}(x), \mathsf{p}(\mathsf{s}(y))))^\sharp = \mathsf{M}(\mathsf{p}(x), \mathsf{p}(\mathsf{s}(y)))$ *is an instance of the left-hand side of (5).*

A CS-TRS is terminating iff there is no infinite chain [1]. As in the non-CS case, the above notion of chains can also be adapted to *innermost* rewriting. Then a CS-TRS is innermost terminating iff there is no infinite innermost chain [4].

Due to the collapsing CS-DPs (and the corresponding definition of "chains"), it is not easy to extend existing techniques for proving absence of infinite chains to CS-DPs. Therefore, we now introduce a new improved definition of CS-DPs.

3 Non-collapsing CS-Dependency Pairs

Ordinary DPs only consider active subterms of right-hand sides. So Rule (1) of Ex. 1 only leads to the DP (5), but not to $\mathsf{M}(x, y) \to \mathsf{M}(\mathsf{p}(x), \mathsf{p}(y))$. However, the inactive subterm $\mathsf{minus}(\mathsf{p}(x), \mathsf{p}(y))$ of the right-hand side of (1) may become active again when applying the rule $\mathsf{if}(\mathsf{true}, x, y) \to x$. Therefore, Def. 2 creates a collapsing DP like (3) whenever a rule $\ell \to r$ has a *migrating variable* x with $r \trianglerighteq_\mu x$, but $\ell \not\rhd_\mu x$. Indeed, when instantiating the *collapse-variable* x in (3) with an instance of the "hidden term" $\mathsf{minus}(\mathsf{p}(x), \mathsf{p}(y))$, one obtains a chain which simulates the rewrite sequence from $\mathsf{minus}(t_1, t_2)$ over $\mathsf{if}(\ldots, \mathsf{minus}(\mathsf{p}(t_1), \mathsf{p}(t_2)), \ldots)$

[1] A refinement is to eliminate DPs where $\ell \rhd_\mu t$, cf. [1,9].

[2] We always assume that different occurrences of DPs are variable-disjoint and consider substitutions whose domains may be infinite.

to $\mathsf{minus}(\mathsf{p}(t_1), \mathsf{p}(t_2))$, cf. Ex. 4. Our main observation is that collapsing DPs are only needed for certain instantiations of the variables. One might be tempted to allow only instantiations of collapse-variables by *hidden terms*.[3]

Definition 5 (Hidden Term). *Let $(\mathcal{R}, \mu)$ be a CS-TRS. We say that t is a* hidden term *if $root(t) \in \mathcal{D}$ and if there exists a rule $\ell \to r \in \mathcal{R}$ with $r \rhd_\mu t$.*

In Ex. 1, the only hidden term is $\mathsf{minus}(\mathsf{p}(x), \mathsf{p}(y))$. But unfortunately, only allowing instantiations of collapse-variables with hidden terms would be unsound.

Example 6. Consider $\mu(\mathsf{g}) = \{1\}$, $\mu(\mathsf{a}) = \mu(\mathsf{b}) = \mu(\mathsf{f}) = \mu(\mathsf{h}) = \varnothing$ and the rules

$$\begin{array}{rl} \mathsf{a} \to \mathsf{f}(\mathsf{g}(\mathsf{b})) & (9) \\ \mathsf{f}(x) \to \mathsf{h}(x) & \end{array} \qquad \begin{array}{rl} \mathsf{h}(x) \to x \\ \mathsf{b} \to \mathsf{a} \end{array}$$

The CS-TRS has the following infinite rewrite sequence:

$$\mathsf{a} \hookrightarrow_{\mathcal{R},\mu} \mathsf{f}(\mathsf{g}(\mathsf{b})) \hookrightarrow_{\mathcal{R},\mu} \underline{\mathsf{h}(\mathsf{g}(\mathsf{b})) \hookrightarrow_{\mathcal{R},\mu} \mathsf{g}(\mathsf{b})} \hookrightarrow_{\mathcal{R},\mu} \mathsf{g}(\mathsf{a}) \hookrightarrow_{\mathcal{R},\mu} \dots$$

We obtain the following CS-DPs according to Def. 2:

$$\begin{array}{rl} \mathsf{A} \to \mathsf{F}(\mathsf{g}(\mathsf{b})) & \\ \mathsf{F}(x) \to \mathsf{H}(x) & \end{array} \qquad \begin{array}{rl} \mathsf{H}(x) \to x \\ \mathsf{B} \to \mathsf{A} \end{array} \tag{10}$$

The only hidden term is b*, obtained from Rule (9). There is also an infinite chain that corresponds to the infinite reduction above. However, here the collapse-variable x in the DP (10) must be instantiated by* $\mathsf{g}(\mathsf{b})$ *and not by the hidden term* b*, cf. the underlined part above. So if one replaced (10) by* $\mathsf{H}(\mathsf{b}) \to \mathsf{b}$*, there would be no infinite chain anymore and one would falsely conclude termination.*

The problem in Ex. 6 is that rewrite rules may add additional symbols like g *above* hidden terms. This can happen if a term $\mathsf{g}(t)$ occurs at an inactive position in a right-hand side and if an instantiation of t could possibly reduce to a term containing a hidden term (i.e., if t has a defined symbol or a variable at an active position). Then we call $\mathsf{g}(\Box)$ a *hiding context*, since it can "hide" a hidden term. Moreover, the composition of hiding contexts is again a hiding context.

Definition 7 (Hiding Context). *Let $(\mathcal{R}, \mu)$ be a CS-TRS. The function symbol f* hides position i *if there is a rule $\ell \to r \in \mathcal{R}$ with $r \rhd_\mu f(r_1, \dots, r_i, \dots, r_n)$, $i \in \mu(f)$, and r_i contains a defined symbol or a variable at an active position. A context C is* hiding *iff $C = \Box$ or C has the form $f(t_1, \dots, t_{i-1}, C', t_{i+1}, \dots, t_n)$ where f hides position i and C' is a hiding context.*

Example 8. In Ex. 6, g *hides position 1 due to Rule (9). So the hiding contexts are* $\Box, \mathsf{g}(\Box), \mathsf{g}(\mathsf{g}(\Box)), \dots$ *In the TRS of Ex. 1,* minus *hides both positions 1 and 2 and* p *hides position 1 due to Rule (1). So the hiding contexts are* $\Box, \mathsf{p}(\Box), \mathsf{minus}(\Box, \Box), \mathsf{p}(\mathsf{p}(\Box)), \mathsf{minus}(\Box, \mathsf{p}(\Box)), \dots$

To remove collapsing DPs $s \to x$, we now restrict ourselves to instantiations of x with terms of the form $C[t]$ where C is a hiding context and t is a hidden term. So in Ex. 6, the variable x in the DP (10) should only be instantiated by b, $\mathsf{g}(\mathsf{b})$,

[3] A similar notion of *hidden symbols* was presented in [2,4], but there one only used these symbols to improve one special termination technique (the *dependency graph*).

$\mathsf{g}(\mathsf{g}(\mathsf{b}))$, etc. To represent these infinitely many instantiations in a finite way, we replace $s \to x$ by new *unhiding* DPs (which "unhide" hidden terms).

Definition 9 (Improved CS-DPs). *For a CS-TRS* $(\mathcal{R}, \mu)$*, if* $\mathsf{DP}_c(\mathcal{R}, \mu) \neq \varnothing$*, we introduce a fresh*[4] unhiding tuple symbol U *and the following* unhiding DPs*:*

- $s \to \mathsf{U}(x)$ *for every* $s \to x \in \mathsf{DP}_c(\mathcal{R}, \mu)$*,*
- $\mathsf{U}(f(x_1, \ldots, x_i, \ldots, x_n)) \to \mathsf{U}(x_i)$ *for every function symbol* f *of any arity* n *and every* $1 \leq i \leq n$ *where* f *hides position* i*, and*
- $\mathsf{U}(t) \to t^\sharp$ *for every hidden term* t*.*

Let $\mathsf{DP}_u(\mathcal{R}, \mu)$ *be the set of all unhiding DPs (where* $\mathsf{DP}_u(\mathcal{R}, \mu) = \varnothing$*, if* $\mathsf{DP}_c(\mathcal{R}, \mu) = \varnothing$*). Then the set of* improved CS-DPs *is* $\mathsf{DP}(\mathcal{R}, \mu) = \mathsf{DP}_o(\mathcal{R}, \mu) \cup \mathsf{DP}_u(\mathcal{R}, \mu)$*.*

Example 10. In Ex. 6, instead of (10) we get the unhiding DPs

$$\mathsf{H}(x) \to \mathsf{U}(x), \qquad \mathsf{U}(\mathsf{g}(x)) \to \mathsf{U}(x), \qquad \mathsf{U}(\mathsf{b}) \to \mathsf{B}.$$

Now there is indeed an infinite chain. In Ex. 7, instead of (3) and (4), we obtain[5]

$$\begin{aligned}
\mathsf{IF}(\mathsf{true}, x, y) &\to \mathsf{U}(x) && (11)\\
\mathsf{IF}(\mathsf{false}, x, y) &\to \mathsf{U}(y) && (12)\\
\mathsf{U}(\mathsf{minus}(\mathsf{p}(x), \mathsf{p}(y))) &\to \mathsf{M}(\mathsf{p}(x), \mathsf{p}(y)) && (13)\\
\mathsf{U}(\mathsf{p}(x)) &\to \mathsf{P}(x) && (14)\\
\mathsf{U}(\mathsf{p}(x)) &\to \mathsf{U}(x) && (15)\\
\mathsf{U}(\mathsf{minus}(x, y)) &\to \mathsf{U}(x) && (16)\\
\mathsf{U}(\mathsf{minus}(x, y)) &\to \mathsf{U}(y) && (17)
\end{aligned}$$

Clearly, the improved CS-DPs are never collapsing. Thus, now the definition of (minimal)[6] *chains* is completely analogous to the one for ordinary rewriting.

Definition 11 (Chain). *Let* $\mathcal{P}$ *and* $\mathcal{R}$ *be TRSs and let* μ *be a replacement map. We extend* μ *to tuple symbols by defining* $\mu(f^\sharp) = \mu(f)$ *for all* $f \in \mathcal{D}$ *and* $\mu(\mathsf{U}) = \varnothing$*.*[7] *A sequence of pairs* $s_1 \to t_1, s_2 \to t_2, \ldots$ *from* $\mathcal{P}$ *is a* $(\mathcal{P}, \mathcal{R}, \mu)$-chain *iff there is a substitution* σ *with* $t_i\sigma \hookrightarrow^*_{\mathcal{R},\mu} s_{i+1}\sigma$ *and* $t_i\sigma$ *is terminating w.r.t.* $(\mathcal{R}, \mu)$ *for all* i*. It is an* innermost $(\mathcal{P}, \mathcal{R}, \mu)$-chain *iff* $t_i\sigma \overset{\mathsf{i}}{\hookrightarrow}^*_{\mathcal{R},\mu} s_{i+1}\sigma$*,* $s_i\sigma$ *is in normal form, and* $t_i\sigma$ *is innermost terminating w.r.t.* $(\mathcal{R}, \mu)$ *for all* i*.*

Our main theorem shows that improved CS-DPs are still sound and complete.

Theorem 12 (Soundness and Completeness of Improved CS-DPs). *A CS-TRS* $(\mathcal{R}, \mu)$ *is terminating iff there is no infinite* $(\mathsf{DP}(\mathcal{R}, \mu), \mathcal{R}, \mu)$*-chain and innermost terminating iff there is no infinite innermost* $(\mathsf{DP}(\mathcal{R}, \mu), \mathcal{R}, \mu)$*-chain.*

Proof. We only prove the theorem for "full" termination. The proof for innermost termination is very similar and can be found in [5].

[4] Alternatively, one could also use different U-symbols for different collapsing DPs.

[5] We omitted the DP $\mathsf{U}(\mathsf{p}(y)) \to \mathsf{P}(y)$ that is "identical" to (14).

[6] Since we only regard *minimal* chains in the following, we included the "minimality requirement" in Def. 11, i.e., we require that all $t_i\sigma$ are (innermost) terminating. As in the DP framework for ordinary rewriting, this restriction to minimal chains is needed for several DP processors (e.g., for the reduction pair processor of Thm. 21).

[7] We define $\mu(\mathsf{U}) = \varnothing$, since the purpose of U is only to remove context around hidden terms. But during this removal, U's argument should not be evaluated.

Soundness

$\mathcal{M}_{\infty,\mu}$ contains all *minimal non-terminating terms*: $t \in \mathcal{M}_{\infty,\mu}$ iff t is non-terminating and every r with $t \rhd_\mu r$ terminates. A term u has the *hiding property* iff

- $u \in \mathcal{M}_{\infty,\mu}$ and
- whenever $u \rhd\!\!\!\!\!\!\cdot_\mu s \trianglerighteq_\mu t'$ for some terms s and t' with $t' \in \mathcal{M}_{\infty,\mu}$, then t' is an instance of a hidden term and $s = C[t']$ for some hiding context C.

We first prove the following claim:

$$\text{Let } u \text{ be a term with the hiding property and let } u \hookrightarrow_{\mathcal{R},\mu} v \trianglerighteq_\mu w \text{ with } w \in \mathcal{M}_{\infty,\mu}. \text{ Then } w \text{ also has the hiding property.} \tag{18}$$

Let $w \rhd\!\!\!\!\!\!\cdot_\mu s \trianglerighteq_\mu t'$ for some terms s and t' with $t' \in \mathcal{M}_{\infty,\mu}$. Clearly, this also implies $v \rhd\!\!\!\!\!\!\cdot_\mu s$. If already $u \rhd s$, then we must have $u \rhd\!\!\!\!\!\!\cdot_\mu s$ due to the minimality of u. Thus, t' is an instance of a hidden term and $s = C[t']$ for a hiding context C, since u has the hiding property. Otherwise, $u \not\rhd s$. There must be a rule $\ell \to r \in \mathcal{R}$, an active context D (i.e., a context where the hole is at an active position), and a substitution δ such that $u = D[\delta(\ell)]$ and $v = D[\delta(r)]$. Clearly, $u \not\rhd s$ implies $\delta(\ell) \not\rhd s$ and $D \not\rhd s$. Hence, $v \rhd\!\!\!\!\!\!\cdot_\mu s$ means $\delta(r) \rhd\!\!\!\!\!\!\cdot_\mu s$. (The root of s cannot be above $\Box$ in D since those positions would be active.) Note that s cannot be at or below a variable position of r, because this would imply $\delta(\ell) \rhd s$. Thus, s is an instance of a non-variable subterm of r that is at an inactive position. So there is a $r' \notin \mathcal{V}$ with $r \rhd\!\!\!\!\!\!\cdot_\mu r'$ and $s = \delta(r')$. Recall that $s \trianglerighteq_\mu t'$, i.e., there is a $p \in \mathcal{P}os^\mu(s)$ with $s|_p = t'$. If p is a non-variable position of r', then $\delta(r'|_p) = t'$ and $r'|_p$ is a subterm with defined root at an active position (since $t' \in \mathcal{M}_{\infty,\mu}$ implies $\mathrm{root}(t') \in \mathcal{D}$). Hence, $r'|_p$ is a hidden term and thus, t' is an instance of a hidden term. Moreover, any instance of the context $C' = r'[\Box]_p$ is hiding. So if we define C to be $\delta(C')$, then $s = \delta(r') = \delta(r')[t']_p = \delta(C')[t'] = C[t']$ for the hiding context C. On the contrary, if p is not a non-variable position of r', then $p = p_1\, p_2$ where $r'|_{p_1}$ is a variable x. Now t' is an active subterm of $\delta(x)$ (more precisely, $\delta(x)|_{p_2} = t'$). Since x also occurs in ℓ, we have $\delta(\ell) \rhd \delta(x)$ and thus $u \rhd \delta(x)$. Due to the minimality of u this implies $u \rhd\!\!\!\!\!\!\cdot_\mu \delta(x)$. Since $u \rhd\!\!\!\!\!\!\cdot_\mu \delta(x) \trianglerighteq_\mu t'$, the hiding property of u implies that t' is an instance of a hidden term and that $\delta(x) = \overline{C}[t']$ for a hiding context $\overline{C}$. Note that since $r'|_{p_1}$ is a variable, the context C' around this variable is also hiding (i.e., $C' = r'[\Box]_{p_1}$). Thus, the context $C = \delta(C')[\overline{C}]$ is hiding as well and $s = \delta(r') = \delta(r')[\delta(x)[t']_{p_2}]_{p_1} = \delta(C')[\overline{C}[t']] = C[t']$.

Proof of Thm. 12 using Claim (18)

If $\mathcal{R}$ is not terminating, then there is a $t \in \mathcal{M}_{\infty,\mu}$ that is minimal w.r.t. $\trianglerighteq$. So there are t, t_i, s_i, t'_{i+1} such that

$$t \xrightarrow{>\varepsilon}{}^*_{\mathcal{R},\mu} t_1 \xrightarrow{\varepsilon}_{\mathcal{R}} s_1 \trianglerighteq_\mu t'_2 \xrightarrow{>\varepsilon}{}^*_{\mathcal{R},\mu} t_2 \xrightarrow{\varepsilon}_{\mathcal{R}} s_2 \trianglerighteq_\mu t'_3 \xrightarrow{>\varepsilon}{}^*_{\mathcal{R},\mu} t_3 \ldots \tag{19}$$

where $t_i, t'_i \in \mathcal{M}_{\infty,\mu}$ and all proper subterms of t (also at *inactive* positions) terminate. Here, "ε" (resp. "$> \varepsilon$") denotes reductions at (resp. strictly below) the root.

Note that (18) implies that all t_i have the hiding property. To see this, we use induction on i. Since t trivially has the hiding property (as it has no non-terminating proper subterms) and all terms in the reduction $t \xrightarrow{>\varepsilon}{}^*_{\mathcal{R},\mu} t_1$ are from $\mathcal{M}_{\infty,\mu}$ (as both $t, t_1 \in \mathcal{M}_{\infty,\mu}$), we conclude that t_1 also has the hiding property by applying (18) repeatedly. In the induction step, if t_{i-1} has the hiding property, then one application of (18) shows that t'_i also has the hiding property. By applying (18) repeatedly, one then also shows that t_i has the hiding property.

Now we show that $t_i^\sharp \to^+_{\mathsf{DP}(\mathcal{R},\mu)} {t'_{i+1}}^\sharp$ and that all terms in the reduction $t_i^\sharp \to^+_{\mathsf{DP}(\mathcal{R},\mu)} {t'_{i+1}}^\sharp$ terminate w.r.t. $(\mathcal{R},\mu)$. As ${t'_{i+1}}^\sharp \xrightarrow{>\varepsilon}{}^*_{\mathcal{R},\mu} t_{i+1}^\sharp$, we get an infinite $(\mathsf{DP}(\mathcal{R},\mu),\mathcal{R},\mu)$-chain.

From (19) we know that there are $\ell_i \to r_i \in \mathcal{R}$ and $p_i \in \mathcal{P}os^\mu(s_i)$ with $t_i = \ell_i\sigma$, $s_i = r_i\sigma$, and $s_i|_{p_i} = r_i\sigma|_{p_i} = t'_{i+1}$ for all i. First let $p_i \in \mathcal{P}os(r_i)$ with $r_i|_{p_i} \notin \mathcal{V}$. Then $\ell_i^\sharp \to (r_i|_{p_i})^\sharp \in \mathsf{DP}_o(\mathcal{R},\mu)$ and $t_i^\sharp = \ell_i^\sharp\sigma \to_{\mathsf{DP}_o(\mathcal{R},\mu)} (r_i|_{p_i})^\sharp\sigma = {t'_{i+1}}^\sharp$. Moreover, as $t_i, t'_{i+1} \in \mathcal{M}_{\infty,\mu}$, the terms $t_i^\sharp$ and ${t'_{i+1}}^\sharp$ are terminating.

Now let p_i be at or below the position of a variable x_i in r_i. By minimality of t_i, x_i only occurs at inactive positions of ℓ_i. Thus, $\ell_i^\sharp \to \mathsf{U}(x_i) \in \mathsf{DP}_u(\mathcal{R},\mu)$ and $r_i = C_i[x_i]$ where C_i is an active context. Recall that $t_i = \ell_i\sigma$ has the hiding property and that $t_i \rhd_\mu \sigma(x_i) \trianglerighteq_\mu t'_{i+1}$. Thus, we have $\sigma(x_i) = C[t'_{i+1}]$ for a hiding context C and moreover, t'_{i+1} is an instance of a hidden term. Hence we obtain:

$$
\begin{array}{lll}
t_i^\sharp & = & \sigma(\ell_i^\sharp) \\
& \to_{\mathsf{DP}_u(\mathcal{R},\mu)} & \mathsf{U}(\sigma(x_i)) \quad \text{since } \ell_i^\sharp \to \mathsf{U}(x_i) \in \mathsf{DP}_u(\mathcal{R},\mu) \\
& = & \mathsf{U}(C[t'_{i+1}]) \quad \text{for a hiding context } C \\
& \to^*_{\mathsf{DP}_u(\mathcal{R},\mu)} & \mathsf{U}(t'_{i+1}) \quad \text{since } \mathsf{U}(C[x]) \to^*_{\mathsf{DP}_u(\mathcal{R},\mu)} \mathsf{U}(x) \text{ for any hiding context } C \\
& \to_{\mathsf{DP}_u(\mathcal{R},\mu)} & {t'_{i+1}}^\sharp \quad \text{since } t'_{i+1} \text{ is an instance of a hidden term and} \\
& & \quad \mathsf{U}(t) \to_{\mathsf{DP}_u(\mathcal{R},\mu)} t^\sharp \text{ for any instance } t \text{ of a hidden term}
\end{array}
$$

All terms in the reduction above are terminating. The reason is that again $t_i, t'_{i+1} \in \mathcal{M}_{\infty,\mu}$ implies that $t_i^\sharp$ and ${t'_{i+1}}^\sharp$ are terminating. Moreover, all terms $\mathsf{U}(\ldots)$ are normal forms since $\mu(\mathsf{U}) = \varnothing$ and since U does not occur in $\mathcal{R}$.

Completeness

Let there be an infinite chain $v_1 \to w_1, v_2 \to w_2, \ldots$ of improved CS-DPs. First, let the chain have an infinite tail consisting only of DPs of the form $\mathsf{U}(f(x_1,\ldots,x_i,\ldots,x_n)) \to \mathsf{U}(x_i)$. Since $\mu(\mathsf{U}) = \varnothing$, there are terms t_i with $\mathsf{U}(t_1) \xrightarrow{\varepsilon}_{\mathsf{DP}(R,\mu)} \mathsf{U}(t_2) \xrightarrow{\varepsilon}_{\mathsf{DP}(R,\mu)} \ldots$ Hence, $t_1 \rhd_\mu t_2 \rhd_\mu \ldots$ which contradicts the well-foundedness of $\rhd_\mu$.

Now we regard the remaining case. Here the chain has infinitely many DPs $v \to w$ with $v = \ell^\sharp$ for a rule $\ell \to r \in \mathcal{R}$. Let $v_i \to w_i$ be such a DP and let $v_j \to w_j$ with $j > i$ be the *next* such DP in the chain. Let σ be the substitution used for the chain. We show that then $v_i^\flat\sigma \hookrightarrow^*_{\mathcal{R},\mu} C[v_j^\flat\sigma]$ for an active context C. Here, $(f^\sharp(t_1,\ldots,t_n))^\flat = f(t_1,\ldots,t_n)$ for all $f \in \mathcal{D}$. Doing this for all such DPs implies that there is an infinite reduction w.r.t. $(\mathcal{R},\mu)$.

If $v_i \to w_i \in \mathsf{DP}_o(R,\mu)$ then the claim is trivial, because then $j = i+1$ and $v_i^\flat\sigma \hookrightarrow_{\mathcal{R},\mu} C[w_i^\flat\sigma] \hookrightarrow^*_{\mathcal{R},\mu} C[v_{i+1}^\flat\sigma]$ for some active context C.

Otherwise, $v_i \to w_i$ has the form $v_i \to \mathsf{U}(x)$. Then $v_i^\flat\sigma \hookrightarrow_{\mathcal{R},\mu} C_1[\sigma(x)]$ for an active context C_1. Moreover, $\mathsf{U}(\sigma(x))$ reduces to $\mathsf{U}(\delta(t))$ for a hidden term t and

a δ by removing hiding contexts. Since hiding contexts are active, $\sigma(x) = C_2[\delta(t)]$ for an active context C_2. Finally, $t^\sharp\delta \xhookrightarrow{>\varepsilon}{}^*_{\mathcal{R},\mu} v_j\sigma$ and thus, $t\delta \xhookrightarrow{>\varepsilon}{}^*_{\mathcal{R},\mu} v_j^\flat\sigma$. By defining $C = C_1[C_2]$, we get $v_i^\flat\sigma \hookrightarrow^+_{\mathcal{R},\mu} C[v_j^\flat\sigma]$. □

4 CS Dependency Pair Framework

By Thm. 12, (innermost) termination of a CS-TRS is equivalent to absence of infinite (innermost) chains. For ordinary rewriting, the *DP framework* is the most recent and powerful collection of methods to prove absence of infinite chains automatically. Due to our new notion of (non-collapsing) CS-DPs, adapting the DP framework to the context-sensitive case now becomes much easier.[8]

In the DP framework, termination techniques operate on *DP problems* instead of TRSs. Def. 13 adapts this notion to context-sensitive rewriting.

Definition 13 (CS-DP Problem and Processor). *A* CS-DP problem *is a tuple $(\mathcal{P}, \mathcal{R}, \mu, e)$, where $\mathcal{P}$ and $\mathcal{R}$ are TRSs, μ is a replacement map, and $e \in \{\mathbf{t}, \mathbf{i}\}$ is a flag that stands for* **t**ermination *or* **i**nnermost termination. *We also call $(\mathcal{P}, \mathcal{R}, \mu)$-chains "$(\mathcal{P}, \mathcal{R}, \mu, \mathbf{t})$-chains" and we call innermost $(\mathcal{P}, \mathcal{R}, \mu)$-chains "$(\mathcal{P}, \mathcal{R}, \mu, \mathbf{i})$-chains". A* CS-DP problem $(\mathcal{P}, \mathcal{R}, \mu, e)$ *is* finite *if there is no infinite $(\mathcal{P}, \mathcal{R}, \mu, e)$-chain.*

A CS-DP processor *is a function Proc that takes a CS-DP problem as input and returns a possibly empty set of CS-DP problems. The processor Proc is* sound *if a CS-DP problem d is finite whenever all problems in $Proc(d)$ are finite.*

For a CS-TRS $(\mathcal{R}, \mu)$, the termination proof starts with the *initial DP problem* $(\mathsf{DP}(\mathcal{R}, \mu), \mathcal{R}, \mu, e)$ where e depends on whether one wants to prove termination or innermost termination. Then sound DP processors are applied repeatedly. If the final processors return empty sets, then (innermost) termination is proved. Since innermost termination is usually easier to show than full termination, one should use $e = \mathbf{i}$ whenever possible. As shown in [12], termination and innermost termination coincide for CS-TRSs $(\mathcal{R}, \mu)$ where $\mathcal{R}$ is *orthogonal* (i.e., left-linear and without critical pairs). So $(DP(\mathcal{R}, \mu), \mathcal{R}, \mu, \mathbf{i})$ would be the initial DP problem for Ex. 1, even when proving full termination. In Sect. 4.1 - 4.3, we recapitulate 3 important DP processors and extend them to context-sensitive rewriting.

4.1 Dependency Graph Processor

The first processor decomposes a DP problem into several sub-problems. To this end, one determines which pairs can follow each other in chains by constructing a *dependency graph*. In contrast to related definitions for collapsing CS-DPs in [1,4], Def. 14 is analogous to the corresponding definition for non-CS rewriting.

Definition 14 (CS-Dependency Graph). *For a CS-DP problem $(\mathcal{P}, \mathcal{R}, \mu, e)$, the nodes of the* $(\mathcal{P}, \mathcal{R}, \mu, e)$-dependency graph *are the pairs of $\mathcal{P}$, and there is an arc from $v \to w$ to $s \to t$ iff $v \to w, s \to t$ is a $(\mathcal{P}, \mathcal{R}, \mu, e)$-chain.*

[8] For this reason, we omitted the proofs in this section and refer to [5] for all proofs.

Example 15. Fig. 1 shows the dependency graph for Ex. 1, for both $e \in \{\mathbf{t}, \mathbf{i}\}$.[9]

A set $\mathcal{P}' \neq \varnothing$ of DPs is a *cycle* if for every $v \to w, s \to t \in \mathcal{P}'$, there is a non-empty path from $v \to w$ to $s \to t$ traversing only pairs of $\mathcal{P}'$. A cycle $\mathcal{P}'$ is a *strongly connected component* ("SCC") if $\mathcal{P}'$ is not a proper subset of another cycle.

One can prove termination separately for each SCC. Thus, the following processor (whose soundness is obvious and completely analogous to the non-context-sensitive case) modularizes termination proofs.

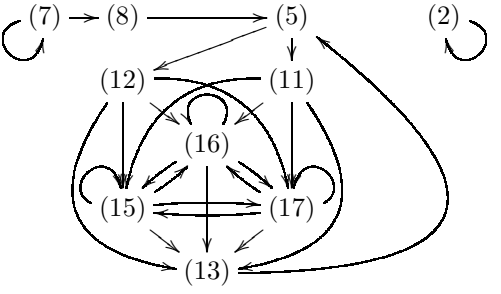


Fig. 1. Dependency graph for Ex. 1

Theorem 16 (CS-Dependency Graph Processor). *For* $d = (\mathcal{P}, \mathcal{R}, \mu, e)$, *let* $Proc(d) = \{(\mathcal{P}_1, \mathcal{R}, \mu, e), \ldots, (\mathcal{P}_n, \mathcal{R}, \mu, e)\}$, *where* $\mathcal{P}_1, \ldots, \mathcal{P}_n$ *are the SCCs of the* $(\mathcal{P}, \mathcal{R}, \mu, e)$*-dependency graph. Then Proc is sound.*

Example 17. The graph in Fig. 1 has the three SCCs $\mathcal{P}_1 = \{(2)\}$, $\mathcal{P}_2 = \{(7)\}$, $\mathcal{P}_3 = \{(5), (11)\text{-}(13), (15)\text{-}(17)\}$. *Thus, the initial DP problem* $(\mathsf{DP}(\mathcal{R}, \mu), \mathcal{R}, \mu, \mathbf{i})$ *is transformed into the new problems* $(\mathcal{P}_1, \mathcal{R}, \mu, \mathbf{i})$, $(\mathcal{P}_2, \mathcal{R}, \mu, \mathbf{i})$, $(\mathcal{P}_3, \mathcal{R}, \mu, \mathbf{i})$.

As in the non-context-sensitive setting, the CS-dependency graph is not computable and thus, one has to use estimations to over-approximate the graph. For example, [1,4] adapted the estimation of [6] that was originally developed for ordinary rewriting: $\textsc{Cap}^{\mu}(t)$ replaces all active subterms of t with defined root symbol by different fresh variables. Multiple occurrences of the same such subterm are also replaced by pairwise different variables. $\textsc{Ren}^{\mu}(t)$ replaces all active occurrences of variables in t by different fresh variables (i.e., no variable occurs at several active positions in $\textsc{Ren}^{\mu}(t)$). So $\textsc{Ren}^{\mu}(\textsc{Cap}^{\mu}(\mathsf{IF}(\mathsf{gt}(y, \mathsf{0}), \mathsf{minus}(\mathsf{p}(x), \mathsf{p}(y)), x))) = \textsc{Ren}^{\mu}(\mathsf{IF}(z, \mathsf{minus}(\mathsf{p}(x), \mathsf{p}(y)), x)) = \mathsf{IF}(z', \mathsf{minus}(\mathsf{p}(x), \mathsf{p}(y)), x)$.

To estimate the CS-dependency graph in the case $e = \mathbf{t}$, one draws an arc from $v \to w$ to $s \to t$ whenever $\textsc{Ren}^{\mu}(\textsc{Cap}^{\mu}(w))$ and s unify.[10] If $e = \mathbf{i}$, then one can modify $\textsc{Cap}^{\mu}$ and $\textsc{Ren}^{\mu}$ by taking into account that instantiated subterms at active positions of the left-hand side must be in normal form, cf. [4]. $\textsc{Cap}^{\mu}_{v}(w)$ is like $\textsc{Cap}^{\mu}(w)$, but the replacement of subterms of w by fresh variables is not done if the subterms also occur at active positions of v. Similarly, $\textsc{Ren}^{\mu}_{v}(w)$ is like $\textsc{Ren}^{\mu}(w)$, but the renaming of variables in w is not done if the variables

[9] To improve readability, we omitted nodes (6) and (14) from the graph. There are arcs from the nodes (8) and (13) to (6) and from all nodes (11), (12), (15), (16), (17) to (14). But (6) and (14) have no outgoing arcs and thus, they are not on any cycle.

[10] Here (and also later in the instantiation processor of Sect. 4.3), we always assume that $v \to w$ and $s \to t$ are renamed apart to be variable-disjoint.

also occur active in v. Now we draw an arc from $v \to w$ to $s \to t$ whenever $\text{REN}_v^\mu(\text{CAP}_v^\mu(w))$ and s unify by an mgu θ where $v\theta$ and $s\theta$ are in normal form.[11]

It turns out that for the TRS of Ex. 1, the resulting estimated dependency graph is identical to the "real" graph in Fig. 1.

4.2 Reduction Pair Processor

There are several processors to simplify DP problems by applying suitable *well-founded orders* (e.g., the *reduction pair processor* [17,21], the *subterm criterion processor* [22], etc.). Due to the absence of collapsing DPs, most of these processors are now straightforward to adapt to the context-sensitive setting. In the following, we present the reduction pair processor with *usable rules*, because it is the only processor whose adaption is more challenging. (The adaption is similar to the one in [4,20] for the CS-DPs of Def. 2.)

To prove that a DP problem is finite, the reduction pair processor generates constraints which should be satisfied by a *μ-reduction pair* $(\gtrsim, \succ)$ [1]. Here, $\gtrsim$ is a stable μ-monotonic quasi-order, $\succ$ is a stable well-founded order, and $\gtrsim$ and $\succ$ are compatible (i.e., $\succ \circ \gtrsim \subseteq \succ$ or $\gtrsim \circ \succ \subseteq \succ$). Here, *$\mu$-monotonicity* means that $s_i \gtrsim t_i$ implies $f(s_1, ..., s_i, ..., s_n) \gtrsim f(s_1, ..., t_i, ..., s_n)$ whenever $i \in \mu(f)$.

For a DP problem $(\mathcal{P}, \mathcal{R}, \mu, e)$, the generated constraints ensure that some rules in $\mathcal{P}$ are strictly decreasing (w.r.t. $\succ$) and all remaining rules in $\mathcal{P}$ and $\mathcal{R}$ are weakly decreasing (w.r.t. $\gtrsim$). Requiring $\ell \gtrsim r$ for all $\ell \to r \in \mathcal{R}$ ensures that in a chain $s_1 \to t_1, s_2 \to t_2, ...$ with $t_i\sigma \hookrightarrow^*_{\mathcal{R},\mu} s_{i+1}\sigma$, we have $t_i\sigma \gtrsim s_{i+1}\sigma$ for all i. Hence, if a reduction pair satisfies the constraints, then one can delete the strictly decreasing pairs from $\mathcal{P}$ as they cannot occur infinitely often in chains.

To improve this idea, it is desirable to require only a weak decrease of *certain* instead of *all* rules. In the non-context-sensitive setting, when proving innermost termination, it is sufficient if just the *usable rules* are weakly decreasing [6]. The same is true when proving full termination, provided that $\gtrsim$ is *$\mathcal{C}_\varepsilon$-compatible*, i.e., $\mathsf{c}(x, y) \gtrsim x$ and $\mathsf{c}(x, y) \gtrsim y$ holds for a fresh function symbol c [17,22].

For a term containing a symbol f, all f-rules are *usable*. Moreover, if the f-rules are usable and f *depends on* h (denoted $f \blacktriangleright_\mathcal{R} h$) then the h-rules are usable as well. Here, $f \blacktriangleright_\mathcal{R} h$ if $f = h$ or if there is a symbol g with $g \blacktriangleright_\mathcal{R} h$ and g occurs in the right-hand side of an f-rule. The usable rules of a DP problem are defined to be the usable rules of the right-hand sides of the DPs.

[11] These estimations can be improved further by adapting existing refinements to the context-sensitive case. However, different to the non-context-sensitive case, for $e = \mathbf{i}$ it is not sufficient to check only for unification of $\text{CAP}_v^\mu(w)$ and s (i.e., renaming variables with REN_v^μ is also needed). This can be seen from the non-innermost terminating CS-TRS $(\mathcal{R}, \mu)$ from [4, Ex. 8] with $\mathcal{R} = \{\mathsf{f}(\mathsf{s}(x), x) \to \mathsf{f}(x, x), \mathsf{a} \to \mathsf{s}(\mathsf{a})\}$ and $\mu(\mathsf{f}) = \{1\}$, $\mu(\mathsf{s}) = \varnothing$. Clearly, $\text{CAP}^\mu_{\mathsf{F}(\mathsf{s}(x),x)}(\mathsf{F}(x, x)) = \mathsf{F}(x, x)$ does not unify with $\mathsf{F}(\mathsf{s}(y), y)$. In contrast, $\text{REN}^\mu_{\mathsf{F}(\mathsf{s}(x),x)}(\text{CAP}^\mu_{\mathsf{F}(\mathsf{s}(x),x)}(\mathsf{F}(x, x))) = \mathsf{F}(x', x)$ unifies with $\mathsf{F}(\mathsf{s}(y), y)$. Thus, without using $\text{REN}^\mu_{\mathsf{F}(\mathsf{s}(x),x)}$ one would conclude that the dependency graph has no cycle and wrongly prove (innermost) termination.

As in [4,20], Def. 18 adapts[12] the concept of usable rules to the CS setting, resulting in $\mathcal{U}^{\blacktriangleright}(\mathcal{P},\mathcal{R},\mu)$. But as shown in [20], for CS rewriting it is also helpful to consider an alternative definition of "dependence" $\rhd_{\mathcal{R},\mu}$ where f also depends on symbols from *left-hand sides* of f-rules. Let $\mathcal{F}^{\mu}(t)$ (resp. $\mathcal{F}^{\not\mu}(t)$) contain all function symbols occurring at active (resp. inactive) positions of a term t.

Definition 18 (CS-Usable Rules). *Let* $Rls(f) = \{\ell \to r \in \mathcal{R} \mid \mathrm{root}(\ell) = f\}$. *For any symbols* f, h *and CS-TRS* $(\mathcal{R},\mu)$, *let* $f \blacktriangleright_{\mathcal{R},\mu} h$ *if* $f = h$ *or if there is a symbol* g *with* $g \blacktriangleright_{\mathcal{R},\mu} h$ *and a rule* $\ell \to r \in Rls(f)$ *with* $g \in \mathcal{F}^{\mu}(r)$. *Let* $f \rhd_{\mathcal{R},\mu} h$ *if* $f = h$ *or if there is a symbol* g *with* $g \rhd_{\mathcal{R},\mu} h$ *and a rule* $\ell \to r \in Rls(f)$ *with* $g \in \mathcal{F}^{\not\mu}(\ell) \cup \mathcal{F}(r)$. *We define two forms of* usable rules*:*

$$\mathcal{U}^{\blacktriangleright}(\mathcal{P},\mathcal{R},\mu) = \bigcup_{s\to t\in\mathcal{P}, f\in\mathcal{F}^{\mu}(t), f\blacktriangleright_{\mathcal{R},\mu} g} Rls(g)$$
$$\mathcal{U}^{\rhd}(\mathcal{P},\mathcal{R},\mu) = \bigcup_{s\to t\in\mathcal{P}, f\in\mathcal{F}^{\not\mu}(s)\cup\mathcal{F}(t), f\rhd_{\mathcal{R},\mu} g} Rls(g) \cup \bigcup_{\ell\to r\in\mathcal{R}, f\in\mathcal{F}^{\not\mu}(r), f\rhd_{\mathcal{R},\mu} g} Rls(g)$$

Example 19. We continue Ex. 17. $\mathcal{U}^{\blacktriangleright}(\mathcal{P}_1,\mathcal{R},\mu) = \varnothing$ *for* $\mathcal{P}_1 = \{(2)\}$, *since there is no defined symbol at an active position in the right-hand side* $\mathsf{GT}(x,y)$ *of (2). For* $\mathcal{P}_2 = \{(7)\}$, $\mathcal{U}^{\blacktriangleright}(\mathcal{P}_2,\mathcal{R},\mu)$ *are the* minus-, if-, *and* gt*-rules, since* minus *occurs at an active position in* $\mathsf{D}(\mathsf{minus}(x,y),\mathsf{s}(y))$ *and* minus *depends on* if *and* gt. *For* $\mathcal{P}_3 = \{(5),(11)\text{-}(13),(15)\text{-}(17)\}$, $\mathcal{U}^{\blacktriangleright}(\mathcal{P}_3,\mathcal{R},\mu)$ *are the* gt- *and* p*-rules, as* gt *and* p *are the only defined symbols at active positions of right-hand sides in* $\mathcal{P}_3$.

In contrast, all $\mathcal{U}^{\rhd}(\mathcal{P}_i,\mathcal{R},\mu)$ *contain all rules except the* div*-rules, as* minus *and* p *are root symbols of hidden terms and* minus *depends on* if *and* gt.

As shown in [4,20], the direct adaption of the usable rules to the context-sensitive case (i.e., $\mathcal{U}^{\blacktriangleright}(\mathcal{P},\mathcal{R},\mu)$) can only be used for *conservative* CS-TRSs (if $e = \mathbf{i}$) resp. for *strongly conservative* CS-TRSs (if $e = \mathbf{t}$).[13] Let $\mathcal{V}^{\mu}(t)$ (resp. $\mathcal{V}^{\not\mu}(t)$) be all variables occurring at active (resp. inactive) positions of a term t.

Definition 20 (Conservative and Strongly Conservative). *A CS-TRS* $(\mathcal{R},\mu)$ *is* conservative *iff* $\mathcal{V}^{\mu}(r) \subseteq \mathcal{V}^{\mu}(\ell)$ *for all rules* $\ell \to r \in \mathcal{R}$. *It is* strongly conservative *iff it is conservative and moreover,* $\mathcal{V}^{\mu}(\ell) \cap \mathcal{V}^{\not\mu}(\ell) = \varnothing$ *and* $\mathcal{V}^{\mu}(r) \cap \mathcal{V}^{\not\mu}(r) = \varnothing$ *for all rules* $\ell \to r \in \mathcal{R}$.

Now we can define the reduction pair processor.

Theorem 21 (CS-Reduction Pair Processor). *Let* $(\succsim,\succ)$ *be a* μ*-reduction pair. For a CS-DP Problem* $d = (\mathcal{P},\mathcal{R},\mu,e)$, *the result of* $Proc(d)$ *is*

- $\{(\mathcal{P} \setminus \succ, \mathcal{R}, \mu, e)\}$, *if* $\mathcal{P} \subseteq (\succ \cup \succsim)$ *and at least one of the following holds:*

[12] The adaptions can also be extended to refined definitions of usable rules [15,17].

[13] The corresponding counterexamples in [4,20] show that these restrictions are still necessary for our new notion of CS-DPs. In cases where one cannot use $\mathcal{U}^{\blacktriangleright}$, one can also attempt a termination proof where one drops the replacement map, i.e., where one regards the ordinary TRS $\mathcal{R}$ instead of the CS-TRS $(\mathcal{R},\mu)$. This may be helpful, since $\mathcal{U}^{\rhd}$ is not necessarily a subset of the non-context-sensitive usable rules, as a function symbol f also $\rhd$-depends on symbols from *left*-hand sides of f-rules.

(i) $\mathcal{U}^{\blacktriangleright}(\mathcal{P},\mathcal{R},\mu) \subseteq \succsim$, $\mathcal{P} \cup \mathcal{U}^{\blacktriangleright}(\mathcal{P},\mathcal{R},\mu)$ *is strongly conservative,* $\succsim$ *is* $\mathcal{C}_\varepsilon$*-compatible*
(ii) $\mathcal{U}^{\blacktriangleright}(\mathcal{P},\mathcal{R},\mu) \subseteq \succsim$, $\mathcal{P} \cup \mathcal{U}^{\blacktriangleright}(\mathcal{P},\mathcal{R},\mu)$ *is conservative,* $e = \mathbf{i}$
(iii) $\mathcal{U}^{\triangleright}(\mathcal{P},\mathcal{R},\mu) \subseteq \succsim$, $\succsim$ *is* $\mathcal{C}_\varepsilon$*-compatible*
(iv) $\mathcal{R} \subseteq \succsim$

- $\{d\}$, *otherwise.*

Then Proc is sound.

Example 22. As $\mathcal{U}^{\blacktriangleright}(\mathcal{P}_1,\mathcal{R},\mu) = \varnothing$ and $\mathcal{P}_1 = \{(2)\}$ *is even strongly conservative, by Thm. 21 (i) or (ii) we only have to orient (2), which already works with the embedding order. So* $(\mathcal{P}_1,\mathcal{R},\mu,\mathbf{i})$ *is transformed to the empty set of DP problems.*

For $\mathcal{P}_2 = \{(7)\}$, $\mathcal{U}^{\blacktriangleright}(\mathcal{P}_2,\mathcal{R},\mu)$ *contains the* if*-rules which are not conservative. Hence, we use Thm. 21 (iii) with a reduction pair based on the following max-polynomial interpretation [10]:* $[\mathsf{D}(x,y)] = [\mathsf{minus}(x,y)] = [\mathsf{p}(x)] = x$, $[\mathsf{s}(x)] = x+1$, $[\mathsf{if}(x,y,z)] = \max(y,z)$, $[\mathsf{0}] = [\mathsf{gt}(x,y)] = [\mathsf{true}] = [\mathsf{false}] = 0$. *Then the DP (7) is strictly decreasing and all rules from* $\mathcal{U}^{\triangleright}(\mathcal{P}_2,\mathcal{R},\mu)$ *are weakly decreasing. Thus, the processor also transforms* $(\mathcal{P}_2,\mathcal{R},\mu,\mathbf{i})$ *to the empty set of DP problems.*

Finally, we regard $\mathcal{P}_3 = \{(5),(11)\text{-}(13),(15)\text{-}(17)\}$ *where we use Thm. 21 (iii) with the interpretation* $[\mathsf{M}(x,y)] = [\mathsf{minus}(x,y)] = x+y+1$, $[\mathsf{IF}(x,y,z)] = [\mathsf{if}(x,y,z)] = \max(y,z)$, $[\mathsf{U}(x)] = [\mathsf{p}(x)] = [\mathsf{s}(x)] = x$, $[\mathsf{0}] = [\mathsf{gt}(x,y)] = [\mathsf{true}] = [\mathsf{false}] = 0$. *Then the DPs (16) and (17) are strictly decreasing, whereas all other DPs from* $\mathcal{P}_3$ *and all rules from* $\mathcal{U}^{\triangleright}(\mathcal{P}_3,\mathcal{R},\mu)$ *are weakly decreasing. So the processor results in the DP problem* $(\{(5),(11)\text{-}(13),(15)\},\mathcal{R},\mu,\mathbf{i})$.

Next we apply $[\mathsf{M}(x,y)] = [\mathsf{minus}(x,y)] = x+1$, $[\mathsf{IF}(x,y,z)] = \max(y,z+1)$, $[\mathsf{if}(x,y,z)] = \max(y,z)$, $[\mathsf{U}(x)] = [\mathsf{p}(x)] = [\mathsf{s}(x)] = x$, $[\mathsf{0}] = [\mathsf{gt}(x,y)] = [\mathsf{true}] = [\mathsf{false}] = 0$. *Now (12) is strictly decreasing and all other remaining DPs and usable rules are weakly decreasing. Removing (12) yields* $(\{(5),(11),(13),(15)\},\mathcal{R},\mu,\mathbf{i})$.

Thm. 21 (iii) and (iv) are a significant improvement over previous reduction pair processors [1,2,4,20] for the CS-DPs from Def. 2. The reason is that all previous CS-reduction pair processors require that the context-sensitive subterm relation is contained in $\succsim$ (i.e., $\trianglerighteq_\mu \subseteq \succsim$) whenever there are collapsing DPs. This is a very hard requirement which destroys one of the main advantages of the DP method (i.e., the possibility to filter away arbitrary arguments).[14] With our new non-collapsing CS-DPs, this requirement is no longer needed.

Example 23. If one requires $\trianglerighteq_\mu \subseteq \succsim$, *then the reduction pair processor would fail for Ex. 1, since then one cannot make the DP (7) strictly decreasing. The reason is that due to* $2 \in \mu(\mathsf{minus})$, $\trianglerighteq_\mu \subseteq \succsim$ *implies* $\mathsf{minus}(x,y) \succsim y$. *So one cannot "filter away" the second argument of* minus. *But then a strict decrease of DP (7) together with* μ*-monotonicity of* $\succsim$ *implies* $\mathsf{D}(\mathsf{s}(x),\mathsf{s}(\mathsf{s}(x))) \succ \mathsf{D}(\mathsf{minus}(x,\mathsf{s}(x)),\mathsf{s}(\mathsf{s}(x))) \succsim \mathsf{D}(\mathsf{s}(x),\mathsf{s}(\mathsf{s}(x)))$, *in contradiction to the well-foundedness of* $\succ$.

[14] Moreover, previous CS-reduction pair processors also require $f(x_1,\ldots,x_n) \succsim f^\sharp(x_1,\ldots,x_n)$ for all $f \in \mathcal{D}$ or $f(x_1,\ldots,x_n) \succ f^\sharp(x_1,\ldots,x_n)$ for all $f \in \mathcal{D}$. This requirement also destroys an important feature of the DP method, i.e., that tuple symbols $f^\sharp$ can be treated independently from the original corresponding symbols f. This feature often simplifies the search for suitable reduction pairs considerably.

4.3 Transforming Context-Sensitive Dependency Pairs

To increase the power of the DP method, there exist several processors to transform a DP into new pairs (e.g., *narrowing*, *rewriting*, *instantiating*, or *forward instantiating* DPs [17]). We now adapt the *instantiation* processor to the context-sensitive setting. Similar adaptions can also be done for the other processors.[15]

The idea of this processor is the following. For a DP $s \to t$, we investigate which DPs $v \to w$ can occur before $s \to t$ in chains. To this end, we use the same estimation as for dependency graphs in Sect. 4.1, i.e., we check whether there is an mgu θ of $\text{REN}^\mu(\text{CAP}^\mu(w))$ and s if $e = \mathbf{t}$ and analogously for $e = \mathbf{i}$.[16] Then we replace $s \to t$ by the new DPs $s\theta \to t\theta$ for all such mgu's θ. This is sound since in any chain $\ldots, v \to w, s \to t, \ldots$ where an instantiation of w reduces to an instantiation of s, one could use the new DP $s\theta \to t\theta$ instead.

Theorem 24 (CS-Instantiation Processor). *Let $\mathcal{P}' = \mathcal{P} \uplus \{s \to t\}$. For $d = (\mathcal{P}', \mathcal{R}, \mu, e)$, let the result of Proc$(d)$ be $(\mathcal{P} \cup \overline{\mathcal{P}}, \mathcal{R}, \mu, e)$ where*

- $\overline{\mathcal{P}} = \{s\theta \to t\theta \mid \theta = mgu(\text{REN}^\mu(\text{CAP}^\mu(w)), s), v \to w \in \mathcal{P}'\}$, *if* $e = \boldsymbol{t}$
- $\overline{\mathcal{P}} = \{s\theta \to t\theta \mid \theta = mgu(\text{REN}^\mu_v(\text{CAP}^\mu_v(w)), s), v \to w \in \mathcal{P}', s\theta, v\theta \text{ normal}\}$, *if* $e = \boldsymbol{i}$

Then Proc is sound.

Example 25. For the TRS of Ex. 1, we still had to solve the problem $(\{(5), (11), (13), (15)\}, \mathcal{R}, \mu, \mathbf{i})$, *cf. Ex. 22. DP (11) has the variable-renamed left-hand side* $\mathsf{IF}(\mathsf{true}, x', y')$. *So the only DP that can occur before (11) in chains is (5) with the right-hand side* $\mathsf{IF}(\mathsf{gt}(y, \mathsf{0}), \mathsf{minus}(\mathsf{p}(x), \mathsf{p}(y)), x)$. *Recall* $\text{REN}^\mu(\text{CAP}^\mu(\mathsf{IF}(\mathsf{gt}(y, \mathsf{0}), \mathsf{minus}(\mathsf{p}(x), \mathsf{p}(y)), x))) = \mathsf{IF}(z', \mathsf{minus}(\mathsf{p}(x), \mathsf{p}(y)), x)$, *cf. Sect. 4.1. So the mgu is* $\theta = [z'/\mathsf{true},\ x'/\mathsf{minus}(\mathsf{p}(x), \mathsf{p}(y)),\ y'/x]$. *Hence, we can replace (11) by*

$$\mathsf{IF}(\mathsf{true}, \mathsf{minus}(\mathsf{p}(x), \mathsf{p}(y)), x) \to \mathsf{U}(\mathsf{minus}(\mathsf{p}(x), \mathsf{p}(y))) \tag{20}$$

Here the CS variant of the instantiation processor is advantageous over the non-CS one which uses CAP *instead of* CAP^μ, *where* CAP *replaces all subterms with defined root (e.g.,* $\mathsf{minus}(\mathsf{p}(x), \mathsf{p}(y))$*) by fresh variables. So the non-CS processor would not help here as it only generates a variable-renamed copy of (11).*

When re-computing the dependency graph, there is no arc from (20) to (15) as $\mu(\mathsf{U}) = \varnothing$. *So the DP problem is decomposed into* $(\{(15)\}, \mathcal{R}, \mu, \mathbf{i})$ *(which is easily solved by the reduction pair processor) and* $(\{(5), (20), (13)\}, \mathcal{R}, \mu, \mathbf{i})$.

Now we apply the reduction pair processor again with the following rational *polynomial interpretation [11]:* $[\mathsf{M}(x, y)] = \frac{3}{2}x + \frac{1}{2}y$, $[\mathsf{minus}(x, y)] = 2x + \frac{1}{2}y$, $[\mathsf{IF}(x, y, z)] = \frac{1}{2}x + y + \frac{1}{2}z$, $[\mathsf{if}(x, y, z)] = \frac{1}{2}x + y + z$, $[\mathsf{U}(x)] = x$, $[\mathsf{p}(x)] = [\mathsf{gt}(x, y)] = \frac{1}{2}x$, $[\mathsf{s}(x)] = 2x + 2$, $[\mathsf{true}] = 1$, $[\mathsf{false}] = [\mathsf{0}] = 0$. *Then (20) is strictly decreasing and can be removed, whereas all other remaining DPs and usable rules*

[15] In the papers on CS-DPs up to now, the only existing adaption of such a processor was the straightforward adaption of the *narrowing* processor in the case $e = \mathbf{t}$, cf. [2]. However, this processor would not help for the TRS of Ex. 1.

[16] The counterexample of [4, Ex. 8] in Footnote 11 again illustrates why REN^μ_v is also needed in the innermost case (whereas this is unnecessary for non-CS rewriting).

are weakly decreasing. A last application of the dependency graph processor then detects that there is no cycle anymore and thus, it returns the empty set of DP problems. Hence, termination of the TRS from Ex. 1 is proved. As shown in our experiments in Sect. 5, this proof can easily be performed automatically.

5 Experiments and Conclusion

We have developed a new notion of context-sensitive dependency pairs which improves significantly over previous notions. There are two main advantages:

(1) **Easier adaption of termination techniques to CS rewriting**
Now CS-DPs are very similar to DPs for ordinary rewriting and consequently, the existing powerful termination techniques from the DP framework can easily be adapted to context-sensitive rewriting. We have demonstrated this with some of the most popular DP processors in Sect. 4. Our adaptions subsume the existing earlier adaptions of the dependency graph [2], of the usable rules [20], and of the modifications for innermost rewriting [4], which were previously developed for the notion of CS-DPs from [1].

(2) **More powerful termination analysis for CS rewriting**
Due to the absence of collapsing CS-DPs, one does not have to impose extra restrictions anymore when extending the DP processors to CS rewriting, cf. Ex. 23. Hence, the power of termination proving is increased substantially.

To substantiate Claim (2), we performed extensive experiments. We implemented our new non-collapsing CS-DPs and all DP processors from this paper in the termination prover AProVE [16].[17] In contrast, the prover MU-TERM [3] uses the collapsing CS-DPs. Moreover, the processors for these CS-DPs are not formulated within the DP framework and thus, they cannot be applied in the same flexible and modular way. While MU-TERM was the most powerful tool for termination analysis of context-sensitive rewriting up to now (as demonstrated by the *International Competition of Termination Tools* 2007 [27]), due to our new notion of CS-DPs, now AProVE is substantially more powerful. For instance, AProVE easily proves termination of our leading example from Ex. 1, whereas MU-TERM fails. Moreover, we tested the tools on all 90 context-sensitive TRSs from the *Termination Problem Data Base* that was used in the competition. We used a time limit of 120 seconds for each example. Then MU-TERM can prove termination of 68 examples, whereas the new version of AProVE proves termination of 78 examples (including all 68 TRSs where MU-TERM is successful).[18] Since 4 examples are known to be non-terminating, at most 8 more of the 90 examples could potentially be detected as terminating. So due to the results of this paper, termination proving of context-sensitive rewriting has now become

[17] We also used the subterm criterion and forward instantiation processors, cf. Sect. 4.

[18] If AProVE is restricted to use exactly the same processors as MU-TERM, then it still succeeds on 74 examples. So its superiority is indeed mainly due to the new CS-DPs which enable an easy adaption of the DP framework to the CS setting.

very powerful. To experiment with our implementation and for details, we refer to http://aprove.informatik.rwth-aachen.de/eval/CS-DPs/.

References

1. Alarcón, B., Gutiérrez, R., Lucas, S.: Context-sensitive dependency pairs. In: Arun-Kumar, S., Garg, N. (eds.) FSTTCS 2006. LNCS, vol. 4337, pp. 297–308. Springer, Heidelberg (2006)
2. Alarcón, B., Gutiérrez, R., Lucas, S.: Improving the context-sensitive dependency graph. In: Proc. PROLE 2006. ENTCS, vol. 188, pp. 91–103 (2007)
3. Alarcón, B., Gutiérrez, R., Iborra, J., Lucas, S.: Proving termination of context-sensitive rewriting with Mu-term. Pr. PROLE 2006. ENTCS, vol. 188, p. 105–115 (2007)
4. Alarcón, B., Lucas, S.: Termination of innermost context-sensitive rewriting using dependency pairs. In: Konev, B., Wolter, F. (eds.) FroCos 2007. LNCS, vol. 4720, pp. 73–87. Springer, Heidelberg (2007)
5. Alarcón, B., Emmes, F., Fuhs, C., Giesl, J., Gutiérrez, R., Lucas, S., Schneider-Kamp, P., Thiemann, R.: Improving context-sensitive dependency pairs. Technical Report AIB-2008-13 (2008), http://aib.informatik.rwth-aachen.de/
6. Arts, T., Giesl, J.: Termination of term rewriting using dependency pairs. Theoretical Computer Science 236, 133–178 (2000)
7. Baader, F., Nipkow, T.: Term Rewriting and All That, Cambridge (1998)
8. Borralleras, C., Lucas, S., Rubio, A.: Recursive path orderings can be context-sensitive. In: Voronkov, A. (ed.) CADE 2002. LNCS, vol. 2392, pp. 314–331. Springer, Heidelberg (2002)
9. Dershowitz, N.: Termination by abstraction. In: Demoen, B., Lifschitz, V. (eds.) ICLP 2004. LNCS, vol. 3132, pp. 1–18. Springer, Heidelberg (2004)
10. Fuhs, C., Giesl, J., Middeldorp, A., Schneider-Kamp, P., Thiemann, R., Zankl, H.: Maximal termination. In: Voronkov, A. (ed.) RTA 2008. LNCS, vol. 5117, pp. 110–125. Springer, Heidelberg (2008)
11. Fuhs, C., Navarro-Marset, R., Otto, C., Giesl, J., Lucas, S., Schneider-Kamp, P.: Search techniques for rational polynomial orders. In: Autexier, S., Campbell, J., Rubio, J., Sorge, V., Suzuki, M., Wiedijk, F. (eds.) AISC 2008, Calculemus 2008, and MKM 2008. LNCS (LNAI), vol. 5144, pp. 109–124. Springer, Heidelberg (2008)
12. Giesl, J., Middeldorp, A.: Innermost termination of context-sensitive rewriting. In: Ito, M., Toyama, M. (eds.) DLT 2002. LNCS, vol. 2450, pp. 231–244. Springer, Heidelberg (2003)
13. Giesl, J., Middeldorp, A.: Transformation techniques for context-sensitive rewrite systems. Journal of Functional Programming 14(4), 379–427 (2004)
14. Giesl, J., Thiemann, R., Schneider-Kamp, P.: The dependency pair framework: Combining techniques for automated termination proofs. In: Baader, F., Voronkov, A. (eds.) LPAR 2004. LNCS, vol. 3452, pp. 301–331. Springer, Heidelberg (2005)
15. Giesl, J., Thiemann, R., Schneider-Kamp, P.: Proving and disproving termination of higher-order functions. In: Gramlich, B. (ed.) FroCos 2005. LNCS (LNAI), vol. 3717, pp. 216–231. Springer, Heidelberg (2005)
16. Giesl, J., Schneider-Kamp, P., Thiemann, R.: AProVE 1.2: Automatic termination proofs in the dependency pair framework. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS, vol. 4130, pp. 281–286. Springer, Heidelberg (2006)

17. Giesl, J., Thiemann, R., Schneider-Kamp, P., Falke, S.: Mechanizing and improving dependency pairs. Journal of Automatic Reasoning 37(3), 155–203 (2006)
18. Gramlich, B.: Generalized sufficient conditions for modular termination of rewriting. Appl. Algebra in Engineering, Comm. and Computing 5, 131–151 (1994)
19. Gramlich, B., Lucas, S.: Simple termination of context-sensitive rewriting. In: Proc. RULE 2002, pp. 29–41. ACM Press, New York (2002)
20. Gutiérrez, R., Lucas, S., Urbain, X.: Usable rules for context-sensitive rewrite systems. In: Voronkov, A. (ed.) RTA 2008. LNCS, vol. 5117, pp. 126–141. Springer, Heidelberg (2008)
21. Hirokawa, N., Middeldorp, A.: Automating the dependency pair method. Information and Computation 199(1,2), 172–199 (2005)
22. Hirokawa, N., Middeldorp, A.: Tyrolean Termination Tool: techniques and features. Information and Computation 205(4), 474–511 (2007)
23. Lucas, S.: Context-sensitive computations in functional and functional logic programs. Journal of Functional and Logic Programming 1998(1), 1–61 (1998)
24. Lucas, S.: Context-sensitive rewriting strategies. Inf. Comp. 178(1), 293–343 (2002)
25. Lucas, S.: Polynomials for proving termination of context-sensitive rewriting. In: Walukiewicz, I. (ed.) FOSSACS 2004. LNCS, vol. 2987, pp. 318–332. Springer, Heidelberg (2004)
26. Lucas, S.: Proving termination of context-sensitive rewriting by transformation. Information and Computation 204(12), 1782–1846 (2006)
27. Marché, C., Zantema, H.: The termination competition. In: Baader, F. (ed.) RTA 2007. LNCS, vol. 4533, pp. 303–313. Springer, Heidelberg (2007)
28. Urbain, X.: Modular & incremental automated termination proofs. Journal of Automated Reasoning 32(4), 315–355 (2004)

Complexity, Graphs, and the Dependency Pair Method*

Nao Hirokawa[1] and Georg Moser[2]

[1] School of Information Science, Japan Advanced Institute of Science and Technology, Japan
hirokawa@jaist.ac.jp
[2] Institute of Computer Science, University of Innsbruck, Austria
georg.moser@uibk.ac.at

Abstract. This paper builds on recent efforts (Hirokawa and Moser, 2008) to exploit the dependency pair method for verifying feasible, i.e., polynomial *runtime complexities* of term rewrite systems automatically. We extend our earlier results by revisiting dependency graphs in the context of complexity analysis. The obtained new results are easy to implement and considerably extend the analytic power of our existing methods. The gain in power is even more significant when compared to existing methods that directly, i.e., without the use of transformations, induce *feasible* runtime complexities. We provide ample numerical data for assessing the viability of the method.

1 Introduction

Term rewriting is a conceptually simple but powerful abstract model of computation that underlies much of declarative programming. *Runtime complexity* is a notion for capturing time complexities of functions defined by a term rewriting system (TRS for short) introduced in [1] (but see also [2,3,4]). In recent research we revisited the basic dependency pair method [5] in order to make it applicable for complexity analysis, cf. [1]. The dependency pair method introduced by Arts and Giesl [5] is one of the most powerful methods in termination analysis. The method enables us to use several powerful techniques including, usable rules, reduction pairs, argument filterings, and dependency graphs. Our main results in [1] show how natural improvements of the dependency pair method, like usable rules, reduction pairs, and argument filterings become applicable in the context of complexity analysis. In this paper, we will extend these recent results further.

The dependency pair method for termination analysis is based on the observation that from an arbitrary non-terminating term one can extract a minimal non-terminating subterm. For that one considers *dependency pairs* that essentially encode recursive calls in a TRS. Notice that with respect to the TRS defined in Example 1 below, one finds 8 such pairs (see Section 4 for further details).

* This research is partly supported by FWF (Austrian Science Fund) project P20133, Grant-in-Aid for Young Scientists 20800022 of the Ministry of Education, Culture, Sports, Science and Technology of Japan, and STARC.

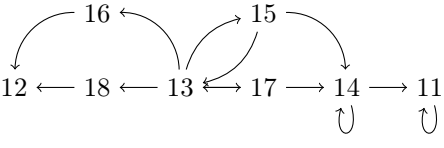
I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 652–666, 2008.

Example 1. Consider the TRS $\mathcal{R}$ which computes the greatest common divisor.[1]

$$
\begin{array}{lrlr}
1\colon & 0 \leqslant y & \to \mathsf{true} & 6\colon \quad \mathsf{gcd}(0,y) \to y \\
2\colon & \mathsf{s}(x) \leqslant 0 & \to \mathsf{false} & 7\colon \quad \mathsf{gcd}(\mathsf{s}(x),0) \to s(x) \\
3\colon & \mathsf{s}(x) \leqslant \mathsf{s}(y)) & \to x \leqslant y & 8\colon \quad \mathsf{gcd}(\mathsf{s}(x),\mathsf{s}(y)) \to \mathsf{if}_{\mathsf{gcd}}(y \leqslant x, \mathsf{s}(x), \mathsf{s}(y)) \\
4\colon & x - 0 & \to x & 9\colon \quad \mathsf{if}_{\mathsf{gcd}}(\mathsf{true}, \mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{gcd}(x-y, \mathsf{s}(y)) \\
5\colon & \mathsf{s}(x) - \mathsf{s}(y) & \to x - y & 10\colon \mathsf{if}_{\mathsf{gcd}}(\mathsf{false}, \mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{gcd}(y-x, \mathsf{s}(x))
\end{array}
$$

A very well-studied refinement of the dependency pair method are *dependency graphs*. To show termination of a TRS, it suffices to guarantee that none of the cycles in $\mathsf{DG}(\mathcal{R})$ [5] can give rise to an infinite rewrite sequence. (Here a *cycle* $\mathcal{C}$ is a nonempty set of dependency pairs of $\mathcal{R}$ such that for every two pairs $s \to t$ and $u \to v$ in $\mathcal{C}$ there exists a nonempty path in $\mathcal{C}$ from $s \to t$ to $u \to v$.) More precisely it suffices to prove for every cycle $\mathcal{C}$ in the dependency graph $\mathsf{DG}(\mathcal{R})$, that there are no $\mathcal{C}$-minimal rewrite sequences (see [7], but also [8,9]). To achieve this one may consider each cycle independently, i.e., for each cycle it suffices to find a reduction pair $(\gtrsim, \succ)$ (cf. Section 2) such that $\mathcal{R} \subseteq \gtrsim$, $\mathcal{C} \subseteq \gtrsim$ and $\mathcal{C} \cap \succ \neq \varnothing$, i.e., at least one dependency pair in $\mathcal{C}$ is strictly decreasing.

Example 2 (continued from Example 1). The *dependency graph* $\mathsf{DG}(\mathcal{R})$, whose nodes are the mentioned 8 dependency pairs, has the following form



This graph contains the maximal cycles $\{11\}$, $\{14\}$, and $\{13, 15, 17\}$, where the latter contains two sub-cycles. As already mentioned, it suffices to consider each of these five cycles individually.

The main contribution of this paper is to extend the dependency graph refinement of the dependency pair method to complexity analysis. This is a challenging task, and we face a couple of difficulties, documented via suitable examples below. To overcome these obstacles we adapt the standard notion of dependency graph suitably and introduce *weak (innermost) dependency graphs*, based on *weak dependency pairs*, which have been studied in [1]. Moreover, we observe that in the context of complexity analysis, it is not enough to focus on the (maximal) cycles of a (weak) dependency graph. Instead, we show how cycle detection is to be replaced by *path detection*, in order to salvage the (standard) technique of dependency graphs for runtime complexity considerations.

The remainder of the paper is organised as follows. After recalling basic notions in Section 2. We recall in Section 3 main results from [1] that will be extended in the sequel. In Section 4 we establish our dependency graph analysis for complexity analysis. Finally, we conclude in Section 5, where we assess the applicability of our method.

[1] This is Example 3.6a in Arts and Giesl's collection of TRSs [6].

2 Preliminaries

We assume familiarity with term rewriting [10,11], but briefly review basic concepts and notations.

Let $\mathcal{V}$ denote a countably infinite set of variables and $\mathcal{F}$ a signature. The set of terms over $\mathcal{F}$ and $\mathcal{V}$ is denoted by $\mathcal{T}(\mathcal{F},\mathcal{V})$ ($\mathcal{T}$ for short). The *root symbol* of a term t is either t itself, if $t \in \mathcal{V}$, or the symbol f, if $t = f(t_1,\dots,t_n)$. The *set of positions* $\mathcal{P}\mathsf{os}(t)$ of a term t is defined as usual. We write $\mathcal{P}\mathsf{os}_{\mathcal{G}}(t) \subseteq \mathcal{P}\mathsf{os}(t)$ for the set of positions of subterms whose root symbol is contained in $\mathcal{G} \subseteq \mathcal{F}$. The *descendants* of a position with respect to a rewrite sequence are defined as usual, cf. [11]. The subterm relation is denoted as $\trianglelefteq$. $\mathcal{V}\mathsf{ar}(t)$ ($\mathcal{F}\mathsf{un}(t)$) denotes the set of variables (functions) occurring in a term t. The *size* $|t|$ of a term is defined as the number of symbols in t. A *term rewrite system* $\mathcal{R}$ over $\mathcal{T}(\mathcal{F},\mathcal{V})$ is a *finite* set of rewrite rules $l \to r$, such that $l \notin \mathcal{V}$ and $\mathcal{V}\mathsf{ar}(l) \supseteq \mathcal{V}\mathsf{ar}(r)$. The smallest rewrite relation that contains $\mathcal{R}$ is denoted by $\to_{\mathcal{R}}$, and its transitive and reflexive closure by $\to^*_{\mathcal{R}}$. We simply write $\to$ for $\to_{\mathcal{R}}$ if $\mathcal{R}$ is clear from context. A term $s \in \mathcal{T}(\mathcal{F},\mathcal{V})$ is called a *normal form* if there is no $t \in \mathcal{T}(\mathcal{F},\mathcal{V})$ such that $s \to t$. The *innermost rewrite relation* $\xrightarrow{\mathsf{i}}_{\mathcal{R}}$ of a TRS $\mathcal{R}$ is defined on terms as follows: $s \xrightarrow{\mathsf{i}}_{\mathcal{R}} t$ if there exists a rewrite rule $l \to r \in \mathcal{R}$, a context C, and a substitution σ such that $s = C[l\sigma]$, $t = C[r\sigma]$, and all proper subterms of $l\sigma$ are normal forms of $\mathcal{R}$. The set of defined symbols is denoted as $\mathcal{D}$, while the constructor symbols are collected in $\mathcal{C}$. We call a term $t = f(t_1,\dots,t_n)$ *basic* if $f \in \mathcal{D}$ and $t_i \in \mathcal{T}(\mathcal{C},\mathcal{V})$ for all $1 \leqslant i \leqslant n$.

We call a TRS *terminating* if no infinite rewrite sequence exists. The n-fold composition of $\to$ is denoted as $\to^n$ and the *derivation length* of a terminating term t with respect to a TRS $\mathcal{R}$ and rewrite relation $\to_{\mathcal{R}}$ is defined as: $\mathsf{dl}(s,\to_{\mathcal{R}}) := \max\{n \mid \exists t\ s \to^n t\}$. Let $\mathcal{R}$ be a TRS and T be a set of terms. The *runtime complexity function with respect to a relation* $\to$ *on* T is defined as follows:

$$\mathsf{rc}(n,T,\to) := \max\{\mathsf{dl}(t,\to) \mid t \in T \text{ and } |t| \leqslant n\}\ .$$

In particular we are interested in the (innermost) runtime complexity with respect to $\to_{\mathcal{R}}$ ($\xrightarrow{\mathsf{i}}_{\mathcal{R}}$) on the set $\mathcal{T}_{\mathsf{b}}$ of all *basic* terms.[2] More precisely, the *runtime complexity function* (with respect to $\mathcal{R}$) is defined as $\mathsf{rc}_{\mathcal{R}}(n) := \mathsf{rc}(n,\mathcal{T}_{\mathsf{b}},\to_{\mathcal{R}})$ and we define the *innermost runtime complexity function* as $\mathsf{rc}^{\mathsf{i}}_{\mathcal{R}}(n) := \mathsf{rc}(n,\mathcal{T}_{\mathsf{b}},\xrightarrow{\mathsf{i}}_{\mathcal{R}})$. Notice that the *derivational complexity function* (with respect to $\mathcal{R}$) becomes definable as follows: $\mathsf{dc}_{\mathcal{R}}(n) := \mathsf{rc}(n,\mathcal{T},\to_{\mathcal{R}})$, where $\mathcal{T}$ denotes the set of *all* terms $\mathcal{T}(\mathcal{F},\mathcal{V})$, compare [12]. We sometimes say the (innermost) runtime complexity of $\mathcal{R}$ is *linear*, *quadratic*, or *polynomial* if $\mathrm{rc}^{(\mathsf{i})}_{\mathcal{R}}$ is bounded by a linear, quadratic, or polynomial function in n, respectively.

A *proper order* is a transitive and irreflexive relation and a *preorder* is a transitive and reflexive relation. A proper order $\succ$ is *well-founded* if there is no infinite decreasing sequence $t_1 \succ t_2 \succ t_3 \cdots$. An $\mathcal{F}$*-algebra* $\mathcal{A}$ consists of a

[2] We can replace $\mathcal{T}_{\mathsf{b}}$ by the set of terms $f(t_1,\dots,t_n)$ with $f \in \mathcal{D}$, whose arguments t_i are in normal form, while keeping all results in this paper.

carrier set A and an interpretation $f_{\mathcal{A}}$ for each function symbol in $\mathcal{F}$. A *well-founded* and *monotone* algebra (*WMA* for short) is a pair $(\mathcal{A}, >)$, where $\mathcal{A}$ is an algebra and $>$ is a well-founded proper order on A such that every $f_{\mathcal{A}}$ is monotone in all arguments. An *assignment* $\alpha\colon \mathcal{V} \to A$ is a function mapping variables to elements in the carrier, and $[\alpha]_{\mathcal{A}}(\cdot)$ denotes the usual evaluation function associated with $\mathcal{A}$. A WMA naturally induces a proper order $>_{\mathcal{A}}$ on terms: $s >_{\mathcal{A}} t$ if $[\alpha]_{\mathcal{A}}(s) > [\alpha]_{\mathcal{A}}(t)$ for all assignments $\alpha\colon \mathcal{V} \to A$. For the reflexive closure $\geqslant$ of $>$, the preorder $\geqslant_{\mathcal{A}}$ is similarly defined. Clearly the proper order $>_{\mathcal{A}}$ is a reduction order, i.e., if $\mathcal{R} \subseteq >_{\mathcal{A}}$, for a TRS $\mathcal{R}$, then we can conclude termination of $\mathcal{R}$. A *rewrite preorder* is a preorder on terms which is closed under contexts and substitutions. A *reduction pair* $(\gtrsim, \succ)$ consists of a rewrite preorder $\gtrsim$ and a compatible well-founded order $\succ$ which is closed under substitutions. Here compatibility means the inclusion $\gtrsim \cdot \succ \cdot \gtrsim \subseteq \succ$. Note that for any WMA $\mathcal{A}$ the pair $(\geqslant_{\mathcal{A}}, >_{\mathcal{A}})$ constitutes a reduction pair.

We call a WMA $\mathcal{A}$ based on the natural numbers $\mathbb{N}$ a *polynomial interpretation*, if all functions $f_{\mathcal{A}}$ are polynomials. A polynomial $P(x_1, \ldots, x_n)$ (over the natural numbers) is called *strongly linear* if $P(x_1, \ldots, x_n) = x_1 + \cdots + x_n + c$ where $c \in \mathbb{N}$. A polynomial interpretation is called *linear restricted* if all constructor symbols are interpreted by strongly linear polynomials and all other function symbols by linear polynomials. If on the other hand the non-constructor symbols are interpreted by quadratic polynomials, the polynomial interpretation is called *quadratic restricted*. Here a polynomial is *quadratic* if it is a sum of monomials of degree at most 2 (see [13]). It is easy to see that if a TRS $\mathcal{R}$ is compatible with a linear or quadratic restricted interpretation, the runtime complexity of $\mathcal{R}$ is linear or quadratic, respectively (see [1] but also [3]). Finally, we introduce a very restrictive class of polynomial interpretations: *strongly linear interpretations* (*SLI* for short). A polynomial interpretation is called *strongly linear* if all functions $f_{\mathcal{A}}$ are interpreted as strongly linear polynomials.

3 Complexity Analysis Based on the Dependency Pair Method

In this section, we recall central definitions and results established in [1]. We kindly refer the reader to [1] for additional examples and underlying intuitions.

We write $C\langle t_1, \ldots, t_n\rangle_X$ to denote $C[t_1, \ldots, t_n]$, whenever $\mathrm{root}(t_i) \in X$ for all $1 \leqslant i \leqslant n$ and C is an n-hole context containing no X-symbols. Let t be a term. We set $t^\sharp := t$ if $t \in \mathcal{V}$, and $t^\sharp := f^\sharp(t_1, \ldots, t_n)$ if $t = f(t_1, \ldots, t_n)$. Here $f^\sharp$ is a new n-ary function symbol called *dependency pair symbol*. For a signature $\mathcal{F}$, we define $\mathcal{F}^\sharp = \mathcal{F} \cup \{f^\sharp \mid f \in \mathcal{F}\}$.

Definition 3. *Let $\mathcal{R}$ be a TRS. If $l \to r \in \mathcal{R}$ and $r = C\langle u_1, \ldots, u_n\rangle_{\mathcal{D}\cup\mathcal{V}}$ then the rewrite rule $l^\sharp \to \mathrm{COM}(u_1^\sharp, \ldots, u_n^\sharp)$ is called a* weak dependency pair *of $\mathcal{R}$. Here* COM *is defined with a fresh n-ary function symbol c (corresponding to $l \to r$) as follows:* $\mathrm{COM}(t_1, \ldots, t_n)$ *is t_1 if $n = 1$, and $c(t_1, \ldots, t_n)$ otherwise. The symbol c is called* compound symbol. *The set of all weak dependency pairs is denoted by* $\mathsf{WDP}(\mathcal{R})$.

Example 4 (continued from Example 1). The set $\mathsf{WDP}(\mathcal{R})$ consists of the next ten weak dependency pairs.

11:	$0 \leqslant^\sharp y \to \mathsf{c}_1$	16:	$\mathsf{gcd}^\sharp(0, y) \to y$
12:	$\mathsf{s}(x) \leqslant^\sharp 0 \to \mathsf{c}_2$	17:	$\mathsf{gcd}^\sharp(\mathsf{s}(x), 0) \to x$
13:	$\mathsf{s}(x) \leqslant^\sharp \mathsf{s}(y) \to x \leqslant^\sharp y$	18:	$\mathsf{gcd}^\sharp(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{if}_{\mathsf{gcd}}{}^\sharp(y \leqslant x, \mathsf{s}(x), \mathsf{s}(y))$
14:	$\mathsf{s}(x) -^\sharp 0 \to x$	19:	$\mathsf{if}_{\mathsf{gcd}}{}^\sharp(\mathsf{true}, \mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{gcd}^\sharp(x - y, \mathsf{s}(y))$
15:	$\mathsf{s}(x) -^\sharp \mathsf{s}(y) \to x -^\sharp y$	20:	$\mathsf{if}_{\mathsf{gcd}}{}^\sharp(\mathsf{false}, \mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{gcd}^\sharp(y - x, \mathsf{s}(x))$

Definition 5. *Let $\mathcal{R}$ be a TRS. If $l \to r \in \mathcal{R}$ and $r = C\langle u_1, \ldots, u_n\rangle_{\mathcal{D}}$ then the rewrite rule $l^\sharp \to \mathrm{COM}(u_1^\sharp, \ldots, u_n^\sharp)$ is called a* weak innermost dependency pair *of $\mathcal{R}$. The set of all weak innermost dependency pairs is denoted by $\mathsf{WIDP}(\mathcal{R})$.*

Definitions 3 and 5 should be compared to the definition of "standard" dependency pairs.

Definition 6 ([5]). *The set $\mathsf{DP}(\mathcal{R})$ of (standard)* dependency pairs *of a TRS $\mathcal{R}$ is defined as $\{l^\sharp \to u^\sharp \mid l \to r \in \mathcal{R}, u \trianglelefteq r, \mathrm{root}(u) \in \mathcal{D}\}$.*

Example 7 (continued from Example 1). As already mentioned in the Introduction, the TRS $\mathcal{R}$ admits 8 (standard) dependency pairs. Notice that the sets $\mathsf{DP}(\mathcal{R})$ and $\mathsf{WDP}(\mathcal{R})$ are incomparable. For example $0 \leqslant^\sharp y \to \mathsf{c}_1 \in \mathsf{WDP}(\mathcal{R}) \setminus \mathsf{DP}(\mathcal{R})$, while $\mathsf{gcd}^\sharp(\mathsf{s}(x), \mathsf{s}(y)) \to y \leqslant^\sharp x \in \mathsf{DP}(\mathcal{R}) \setminus \mathsf{WDP}(\mathcal{R})$.

We write $f \rhd_{\mathrm{d}} g$ if there exists a rewrite rule $l \to r \in \mathcal{R}$ such that $f = \mathrm{root}(l)$ and g is a defined symbol in $\mathcal{F}\mathsf{un}(r)$. For a set $\mathcal{G}$ of defined symbols we denote by $\mathcal{R}{\restriction}\mathcal{G}$ the set of rewrite rules $l \to r \in \mathcal{R}$ with $\mathrm{root}(l) \in \mathcal{G}$. The set $\mathcal{U}(t)$ of usable rules of a term t is defined as $\mathcal{R}{\restriction}\{g \mid f \rhd_{\mathrm{d}}^* g \text{ for some } f \in \mathcal{F}\mathsf{un}(t)\}$. Finally, if $\mathcal{P}$ is a set of (weak or weak innermost) dependency pairs then $\mathcal{U}(\mathcal{P}) = \bigcup_{l\to r\in\mathcal{P}} \mathcal{U}(r)$.

Proposition 8 ([1]). *Let $\mathcal{R}$ be a TRS and let $t \in \mathcal{T}_{\mathsf{b}}$. If t is terminating with respect to $\to$ then $\mathsf{dl}(t, \to) \leqslant \mathsf{dl}(t^\sharp, \to_{\mathcal{U}(\mathcal{P})\cup\mathcal{P}})$, where $\to$ denotes $\to_{\mathcal{R}}$ or $\xrightarrow{i}_{\mathcal{R}}$ depending on whether $\mathcal{P} = \mathsf{WDP}(\mathcal{R})$ or $\mathcal{P} = \mathsf{WIDP}(\mathcal{R})$.*

We recall the notion of *relative rewriting* [11]. Let $\mathcal{R}$ and $\mathcal{S}$ be TRSs. We write $\to_{\mathcal{R}/\mathcal{S}}$ for $\to_{\mathcal{S}}^* \cdot \to_{\mathcal{R}} \cdot \to_{\mathcal{S}}^*$ and we call $\to_{\mathcal{R}/\mathcal{S}}$ the *relative rewrite relation* of $\mathcal{R}$ over $\mathcal{S}$. (Note that $\to_{\mathcal{R}/\mathcal{S}} = \to_{\mathcal{R}}$, if $\mathcal{S} = \varnothing$.) Let $\mathcal{A}$ denote a strongly linear interpretation.

Proposition 9 ([1]). *Let $\mathcal{R}$ and $\mathcal{S}$ be TRSs, and $\mathcal{A}$ an SLI compatible with $\mathcal{S}$. There exist constants K and L, depending only on $\mathcal{R}$ and $\mathcal{A}$, such that $\mathsf{dl}(t, \to_{\mathcal{R}\cup\mathcal{S}}) \leqslant K \cdot \mathsf{dl}(t, \to_{\mathcal{R}/\mathcal{S}}) + L \cdot |t|$ for all terminating terms t on $\mathcal{R} \cup \mathcal{S}$.*

We need some further definitions. Let $\mathcal{R}$ be a TRS, let $\mathcal{P}$ a set of weak or weak innermost dependency pairs of $\mathcal{R}$ and let G denote a mapping associating a term (over $\mathcal{F}^\sharp$ and $\mathcal{V}$) and a proper order $\succ$ with a natural number. An order $\succ$ on terms is G-*collapsible* for a TRS $\mathcal{R}$ if $s \to_{\mathcal{P}\cup\mathcal{U}(\mathcal{P})} t$ and $s \succ t$ implies

$\mathsf{G}(s,\succ) > \mathsf{G}(t,\succ)$. An order $\succ$ is *collapsible* for a TRS $\mathcal{R}$, if there is a mapping G such that $\succ$ is G-collapsible for $\mathcal{R}$.[3]

We write $\mathcal{T}_{\mathsf{b}}^{\sharp}$ for $\{t^{\sharp} \mid t \in \mathcal{T}_{\mathsf{b}}\}$. The set $\mathcal{T}_{\mathsf{c}}^{\sharp}$ is inductively defined as follows (i) $\mathcal{T}^{\sharp} \cup \mathcal{T} \subseteq \mathcal{T}_{\mathsf{c}}^{\sharp}$, where $\mathcal{T}^{\sharp} = \{t^{\sharp} \mid t \in \mathcal{T}\}$. And (ii) $c(t_1,\dots,t_n) \in \mathcal{T}_{\mathsf{c}}^{\sharp}$, whenever $t_1,\dots,t_n \in \mathcal{T}_{\mathsf{c}}^{\sharp}$ and c a compound symbol. A proper order $\succ$ on $\mathcal{T}_{\mathsf{c}}^{\sharp}$ is called *safe* if $c(s_1,\dots,s_i,\dots,s_n) \succ c(s_1,\dots,t,\dots,s_n)$ for all n-ary compound symbols c and all terms $s_1,\dots,s_n,t$ with $s_i \succ t$. A reduction pair $(\gtrsim,\succ)$ is called *collapsible* for a TRS $\mathcal{R}$ if $\succ$ is collapsible for $\mathcal{R}$. It is called *safe* if the well-founded order $\succ$ is safe. In order to construct safe reduction pairs one may use *safe algebras*, i.e., weakly monotone well-founded algebras $(\mathcal{A},\succ)$ such that the interpretations of compound symbols are strictly monotone with respect to $\succ$. It is easy to see that if $(\mathcal{A},>)$ is a safe algebra then $(\geqslant_{\mathcal{A}},>_{\mathcal{A}})$ is a safe reduction pair.

Proposition 10 ([1]). *Let $\mathcal{R}$ be a TRS, let $\mathcal{A}$ an SLI, let $\mathcal{P}$ be the set of weak or weak innermost dependency pairs, and let $(\gtrsim,\succ)$ be a safe and G-collapsible reduction pair such that $\mathcal{U}(\mathcal{P}) \subseteq \gtrsim$ and $\mathcal{P} \subseteq \succ$. If in addition $\mathcal{U}(\mathcal{P}) \subseteq >_{\mathcal{A}}$ then for any $t \in \mathcal{T}_{\mathsf{b}}$, there exist constants K and L (depending only on $\mathcal{R}$ and $\mathcal{A}$) such that $\mathsf{dl}(t,\to) \leqslant K \cdot \mathsf{G}(t^{\sharp},\succ) + L \cdot |t|$. Here $\to$ denotes $\to_{\mathcal{R}}$ or $\xrightarrow{i}_{\mathcal{R}}$ depending on whether $\mathcal{P} = \mathsf{WDP}(\mathcal{R})$ or $\mathcal{P} = \mathsf{WIDP}(\mathcal{R})$.*

Suppose the assertions of the proposition are met and there exists a polynomial p such that $\mathsf{G}(t^{\sharp},\succ) \leqslant p(|t|)$ holds. Then, as an easy corollary to Proposition 10, we observe that the runtime complexity induced by $\mathcal{R}$ is majorised by p.

4 Dependency Graphs

In this section, we study a natural refinement of the dependency pair method, namely *dependency graphs* (see [5,7,14,8]) in the context of complexity analysis. We start with a brief motivation. Let $\mathcal{R}$ be a TRS, let $\mathcal{P}$ denote a set of weak or weak innermost dependency pairs and let $(s_i)_{i=0,\dots,n}$ denote a maximal derivation D with respect to $\mathcal{R}$ with $s_0 \in \mathcal{T}_{\mathsf{b}}$. In order to estimate the length ℓ of this derivation it suffices to estimate the length of the derivation $t_0 \to^*_{\mathcal{U}(\mathcal{P})\cup\mathcal{P}} t_n$, where $t_0 = s_0^{\sharp} \in \mathcal{T}_{\mathsf{b}}^{\sharp}$, cf. Proposition 8. If we suppose compatibility of $\mathcal{U}(\mathcal{P})$ with a strongly linear interpretation $\mathcal{A}$, we may estimate the derivation length ℓ by finding *one* (safe and collapsible) reduction pair $(\gtrsim,\succ)$ such that $\mathcal{U}(\mathcal{P}) \subseteq \gtrsim$ and $\mathcal{P} \subseteq \succ$ holds, cf. Proposition 10. On the other hand in termination analysis—as already mentioned in the Introduction—it suffices to guarantee that for any cycle $\mathcal{C}$ in the dependency graph $\mathsf{DG}(\mathcal{R})$, there are no $\mathcal{C}$-minimal rewrite sequences, cf. [7]. Hence, we strive to extend this idea to complexity analysis.

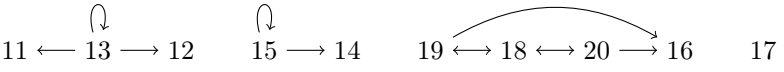
4.1 From Cycle Analysis to Path Detection

Let us recall the definition of a dependency graph and extend it suitably to weak and weak innermost dependency pairs.

[3] Note that most reduction orders are collapsible. E.g. if $\mathcal{A}$ is a polynomial interpretation then $>_{\mathcal{A}}$ is collapsible, as one may take any α and set $\mathsf{G}(t,>_{\mathcal{A}}) := [\alpha]_{\mathcal{A}}(t)$.

Definition 11. *Let $\mathcal{R}$ be a TRS over a signature $\mathcal{F}$ and let $\mathcal{P}$ be the set of weak, weak innermost, or (standard) dependency pairs. The nodes of the* weak dependency graph $\mathsf{WDG}(\mathcal{R})$*,* weak innermost dependency graph $\mathsf{WIDG}(\mathcal{R})$*, or* dependency graph $\mathsf{DG}(\mathcal{R})$ *are the elements of $\mathcal{P}$ and there is an arrow from $s \to t$ to $u \to v$ if and only if there exist a context C and substitutions $\sigma, \tau \colon \mathcal{V} \to \mathcal{T}(\mathcal{F}, \mathcal{V})$ such that $t\sigma \to^* C[u\tau]$, where $\to$ denotes $\to_{\mathcal{R}}$ or $\xrightarrow{\mathsf{i}}_{\mathcal{R}}$ depending on whether $\mathcal{P} = \mathsf{WDP}(\mathcal{R})$, $\mathcal{P} = \mathsf{DP}(\mathcal{R})$ or $\mathcal{P} = \mathsf{WIDP}(\mathcal{R})$, respectively.*

Example 12 (continued from Example 4). The weak dependency graph $\mathsf{WDG}(\mathcal{R})$ has the following form.



We recall a theorem on the dependency graph refinement in conjunction with usable rules and innermost rewriting (see [7], but also [15]). Similar results hold in the context of full rewriting, see [9,8].

Theorem 13 ([7]). *A TRS $\mathcal{R}$ is innermost terminating if for every maximal cycle $\mathcal{C}$ in the dependency graph $\mathsf{DG}(\mathcal{R})$ there exists a reduction pair $(\gtrsim, \succ)$ such that $\mathcal{U}(\mathcal{C}) \subseteq \gtrsim$ and $\mathcal{C} \subseteq \succ$.*

The following example shows that we cannot directly employ Theorem 13 in the realm of complexity analysis. Even though in this setting we can restrict our attention to a specific strategy: innermost rewriting.

Example 14. Consider the TRS $\mathcal{R}_{\mathsf{exp}}$

$$\begin{aligned} \mathsf{exp}(0) &\to \mathsf{s}(0) & \mathsf{d}(0) &\to 0 \\ \mathsf{exp}(\mathsf{r}(x)) &\to \mathsf{d}(\mathsf{exp}(x)) & \mathsf{d}(\mathsf{s}(x)) &\to \mathsf{s}(\mathsf{s}(\mathsf{d}(x))) \end{aligned}$$

$\mathsf{DP}(\mathcal{R}_{\mathsf{exp}})$ consists of three pairs: $1\colon \mathsf{exp}^\sharp(\mathsf{r}(x)) \to \mathsf{d}^\sharp(\mathsf{exp}(x))$, $2\colon \mathsf{exp}^\sharp(\mathsf{r}(x)) \to \mathsf{exp}^\sharp(x)$, and $3\colon \mathsf{d}^\sharp(\mathsf{s}(x)) \to \mathsf{d}^\sharp(x)$. Hence the dependency graph $\mathsf{DG}(\mathcal{R}_{\mathsf{exp}})$ contains two maximal cycles: $\{2\}$ and $\{3\}$. It is easy to see how to define two reduction pairs $(\geqslant_{\mathcal{A}}, >_{\mathcal{A}})$ and $(\geqslant_{\mathcal{B}}, >_{\mathcal{B}})$ such that the conditions of the theorem are fulfilled. For that it suffices to define interpretations $\mathcal{A}$ and $\mathcal{B}$, respectively. Because one can find suitable linear restricted ones for $\mathcal{A}$ and $\mathcal{B}$, compatibility with these interpretations apparently induces *linear* runtime complexity of $\mathcal{R}_{\mathsf{exp}}$, cf. [3,1] (even for full rewriting). However, we must not conclude linear innermost runtime complexity for $\mathcal{R}_{\mathsf{exp}}$ in this setting, as $\mathcal{R}_{\mathsf{exp}}$ formalises the exponentiation function and setting $t_n = \mathsf{exp}(\mathsf{r}^n(0))$ we obtain $\mathsf{dl}(t_n, \xrightarrow{\mathsf{i}}_{\mathcal{R}}) \geqslant 2^n$ for each $n \geqslant 0$. Thus the innermost runtime complexity of $\mathcal{R}_{\mathsf{exp}}$ is exponential.

Note that the problem exemplified by Example 14 cannot be circumvented by replacing the dependency graph employed in Theorem 13 with weak (innermost) dependency graphs. Furthermore, observe that while Proposition 8 allows us to replace in Example 14 the innermost rewrite relation $\xrightarrow{\mathsf{i}}_{\mathcal{R}}$ by the (sometimes simpler) rewrite relation $\xrightarrow{\mathsf{i}}_{\mathcal{U}(\mathsf{DP}(\mathcal{R}))\cup\mathsf{DP}(\mathcal{R})}$, this is of no help: The exponential

length of $t_n^\sharp$ in Example 14 with respect to $\mathcal{U}(\mathsf{DP}(\mathcal{R})) \cup \mathsf{DP}(\mathcal{R})$ is *not* due to the cycles $\{2\}$ or $\{3\}$, but achieved through the non-cyclic pair 1 and its usable rules. These observations are cast into Definition 15, below.

A graph is called *strongly connected* if any node is connected with every other node by a path. A *strongly connected component* (*SCC* for short) is a maximal strongly connected subgraph.[4]

Definition 15. *Let $\mathcal{G}$ be a graph, let $\equiv$ denote the equivalence relation induced by SCCs, and let $\mathcal{P}$ be a SCC in $\mathcal{G}$. The set of all source nodes in $\mathcal{G}/_\equiv$ is denoted by* Src. *Let $l \to r$ be a dependency pair in G, let $\mathcal{K} \in \mathcal{G}/_\equiv$ and let $\mathcal{C}$ denote the SCC represented by $\mathcal{K}$. Then we write $l \to r \in \mathcal{K}$ if $l \to r \in \mathcal{C}$.*

Example 16 (Continued from Example 12). There are 8 SCCs in $\mathsf{WDG}(\mathcal{R})$, almost all except $\{18, 19, 20\}$ being trivial. Hence the graph $\mathsf{WDG}(\mathcal{R})/_\equiv$ has the following form and $\mathsf{Src} = \{\{13\}, \{15\}, \{17\}, \{18, 19, 20\}\}$.

$$11 \longleftarrow 13 \longrightarrow 12 \qquad 15 \longrightarrow 14 \qquad \{18,19,20\} \longrightarrow 16 \qquad 17$$

4.2 Refinement Based on Path Detection

We re-consider the motivating derivation D:

$$t_0 \to_{\mathcal{U}(\mathcal{P})\cup\mathcal{P}} t_1 \to_{\mathcal{U}(\mathcal{P})\cup\mathcal{P}} \cdots \to_{\mathcal{U}(\mathcal{P})\cup\mathcal{P}} t_n \;, \tag{1}$$

where $t_0 \in \mathcal{T}_\mathsf{b}^\sharp$. To simplify the exposition, we set $\mathcal{P} = \mathsf{WDP}(\mathcal{R})$ and $\mathcal{G} = \mathsf{WDG}(\mathcal{R})$. Momentarily we assume that all compound symbol are of arity 0, as is for instance the case in Example 4. Above we asserted that there exists an SLI $\mathcal{A}$ such that $\mathcal{U}(\mathcal{P}) \subseteq >_\mathcal{A}$. Hence Proposition 9 is applicable. Thus, to estimate the length of the derivation (1) it suffices to consider the following relative rewriting derivation:

$$t_0 \to_{\mathcal{P}/\mathcal{U}(\mathcal{P})} t_1 \to_{\mathcal{P}/\mathcal{U}(\mathcal{P})} \cdots \to_{\mathcal{P}/\mathcal{U}(\mathcal{P})} t_n \;. \tag{2}$$

Exploiting the given assumptions, it is not difficult to see that derivation (2) is representable as follows:

$$t_0 \to^{\ell_1}_{\mathcal{P}_1/\mathcal{U}(\mathcal{P}_1)} t_{\ell_1} \to^{\ell_2}_{\mathcal{P}_2/\mathcal{U}(\mathcal{P}_1)\cup\mathcal{U}(\mathcal{P}_2)} \cdots \to^{\ell_m}_{\mathcal{P}_m/\mathcal{U}(\mathcal{P}_1)\cup\cdots\cup\mathcal{U}(\mathcal{P}_m)} t_n \tag{3}$$

where, $(\mathcal{P}_1, \ldots, \mathcal{P}_m)$ is a *path* in $\mathcal{G}/_\equiv$ with $\mathcal{P}_1 \in \mathsf{Src}$. Since the length ℓ of the pictured $\to_{\mathcal{P}/\mathcal{U}(\mathcal{P})}$-rewrite sequence equals $\ell_1 + \cdots + \ell_m$, this suggests that we can estimate each ℓ_j ($j \in \{1, \ldots, m\}$) independently. We assume the existence of a family of SLIs $\mathcal{B}_j$ ($j \in \{1, \ldots, m\}$) such that $\mathcal{U}(\mathcal{P}_1) \cup \cdots \cup \mathcal{U}(\mathcal{P}_j) \subseteq \geqslant_{\mathcal{B}_j}$ and $\mathcal{P}_j \subseteq >_{\mathcal{B}_j}$ holds for every i. From this we can conclude $\ell_j = \mathsf{O}(|t_{\ell_j}|)$ for all $j \in \{1, \ldots, m\}$. The next step is to estimate each ℓ_j by a function (preferable a polynomial) in $|t_0|$. As each of the WMAs $\mathcal{B}_j$ is assumed to be strongly linear, we can even conclude $[\alpha_0]_{\mathcal{B}_j}(t_{\ell_j}) = \Omega(|t_{\ell_j}|)$.(Here α_0 denotes the assignment

[4] Note that in the literature SCCs are sometimes defined as *maximal cycles*. This alternative definition is of limited use in our context.

mapping any variable to 0.) In sum, we obtain for each $j \in \{1,\dots,m\}$, the existence of a constant c_j such that $|t_{\ell_j}| \leqslant c_j \cdot |t_0|$ and thus there exists a linear polynomial $p(x)$ such that $\ell_j \leqslant p(|t_0|)$. However, some care is necessary in assessing this observation: Note that the given argument cannot be used to deduce *polynomial* runtime complexity, if we weaken the assumption that the algebras $\mathcal{B}_j$ are strongly linear only slightly. Hence, we replace the direct application of Proposition 9 as follows.

Lemma 17. *$n \leqslant \mathsf{dl}(s, \to_{\mathcal{R}_2/\mathcal{S}_1 \cup \mathcal{S}_2})$ whenever $s \to_{\mathcal{S}_1}{}^* \cdot \to_{\mathcal{R}_2/\mathcal{S}_2}{}^n u$.*

Proof. Straightforward. □

We lift the assumption that all compound symbols are of arity at most 0. Perhaps surprisingly this generalisation complicates the matter considerably. First a *maximal* derivation need no longer be of the form given in (3) which is exemplified by Example 18 below.

Example 18. Consider the TRS $\mathcal{R} = \{\mathsf{f}(0) \to \mathsf{leaf}, \mathsf{f}(\mathsf{s}(x)) \to \mathsf{branch}(\mathsf{f}(x), \mathsf{f}(x))\}$ The set $\mathsf{WDP}(\mathcal{R})$ consists of the two weak dependency pairs: $1\colon \mathsf{f}^\sharp(0) \to \mathsf{c}_1$ and $2\colon \mathsf{f}^\sharp(\mathsf{s}(x)) \to \mathsf{c}_2(\mathsf{f}^\sharp(x), \mathsf{f}^\sharp(x))$. Hence the weak dependency graph $\mathsf{WDG}(\mathcal{R})$ contains 2 SCCs: $\{2\}$ and $\{1\}$. Clearly $\mathsf{Src} = \{\{2\}\}$. Let $t_n = \mathsf{f}^\sharp(\mathsf{s}^n(0))$. Consider the following sequence:

$$\begin{aligned} t_2 &\to_{\{2\}}{}^2 \mathsf{c}_2(\mathsf{c}_2(t_0,t_0),t_1) \to_{\{1\}} \mathsf{c}_2(\mathsf{c}_2(\mathsf{c}_1,t_0),t_1) \\ &\to_{\{2\}} \mathsf{c}_2(\mathsf{c}_2(\mathsf{c}_1,t_0),\mathsf{c}_2(t_0,t_0)) \to_{\{1\}}{}^3 \mathsf{c}_2(\mathsf{c}_2(\mathsf{c}_1,\mathsf{c}_1),\mathsf{c}_2(\mathsf{c}_1,\mathsf{c}_1)) \,. \end{aligned}$$

This derivation does not have the form (3), because it is based on the sequence $(\{2\},\{1\},\{2\},\{1\})$, which is not a path in $\mathsf{WDG}(\mathcal{R})/_{\equiv}$.

Notice that the derivation in Example 18 can be reordered (without affecting its length) such that the derivation becomes based on a path. Still, not every derivation can be abstracted to a path. Consider a maximal (with respect to subset inclusion) component of $\mathsf{WDG}(\mathcal{R})/_{\equiv}$. Clearly this component forms a directed acyclic graph $\mathcal{G}$, and without loss of generality we can conceive $\mathcal{G}$ as a tree T with root in Src. Suppose further that T is not degenerated to a branch. Then a given derivation may only be abstractable by different paths in T, as exemplified by Example 19.

Example 19. Consider the TRS $\mathcal{R} = \{\mathsf{f} \to \mathsf{c}(\mathsf{g},\mathsf{h}), \mathsf{g} \to \mathsf{a}, \mathsf{h} \to \mathsf{a}\}$. Thus $\mathsf{WDP}(\mathcal{R})$ consists of three dependency pairs: $1\colon \mathsf{f}^\sharp \to \mathsf{c}_1(\mathsf{g}^\sharp,\mathsf{h}^\sharp)$, $2\colon \mathsf{g}^\sharp \to \mathsf{c}_2$, and $3\colon \mathsf{h}^\sharp \to \mathsf{c}_3$. Let $\mathcal{P} := \mathsf{WDP}(\mathcal{R})$, then Then clearly $\mathcal{P} = \mathsf{WDG}(\mathcal{R}) = \mathsf{WDG}(\mathcal{R})/_{\equiv}$. Consider the following derivation

$$\mathsf{f}^\sharp \to_{\mathcal{P}} \mathsf{c}_1(\mathsf{g}^\sharp,\mathsf{h}^\sharp) \to_{\mathcal{P}} \mathsf{c}_1(\mathsf{c}_2,\mathsf{h}^\sharp) \to_{\mathcal{P}} \mathsf{c}_1(\mathsf{c}_2,\mathsf{c}_3) \,.$$

This derivation is composed from the paths $(\{1\},\{2\})$ and $(\{1\},\{3\})$.

Fortunately, we can circumvent these obstacles. Let $\mathcal{P}$ denote the set of weak or weak innermost dependency pairs of a TRS $\mathcal{R}$. We make the following easy observation.

Lemma 20. *Let $\mathcal{G}$ denote a weak or weak innermost dependency graph. Let $\mathcal{C} \subseteq \mathcal{G}$ and let $D\colon s \to^*_{\mathcal{C}/\mathcal{U}(\mathcal{P})} t$ denote a derivation based on $\mathcal{C}$ with $s \in \mathcal{T}^\sharp_\mathsf{c}$. Then D has the following form: $s = s_0 \to_{\mathcal{C}/\mathcal{U}(\mathcal{P})} s_1 \to_{\mathcal{C}/\mathcal{U}(\mathcal{P})} \cdots \to_{\mathcal{C}/\mathcal{U}(\mathcal{P})} s_n = t$ where each $s_i \in \mathcal{T}^\sharp_\mathsf{c}$.*

Proof. It is easy to see that D has the presented form and that for each $i \in \{0, \ldots, n\}$ there exists a context C such that $s_i = C[u^\sharp_1, \ldots, u^\sharp_r]$ and C consists of compound symbols only. This establishes the lemma. □

Motivated by Example 18 we observe that a weak (innermost) dependency pair containing an m-ary ($m > 1$) compound symbol can only induces m *independent* derivations. Hence, we can reorder derivations to achieve the structure of derivation (3). This is formally proven via the next two lemmas.

Lemma 21. *Let $\mathcal{G}$ denote a weak or weak innermost dependency graph and let $\mathcal{K}$ and $\mathcal{L}$ denote two different nodes in $\mathcal{G}/_\equiv$ such that there is no edge from $\mathcal{K}$ to $\mathcal{L}$. Let $s_0 \in \mathcal{T}^\sharp_\mathsf{c}$ and suppose the existence of a derivation D of the following form: $s_0 \to^n_{\mathcal{K}/\mathcal{U}(\mathcal{P})} s_n \to^*_{\mathcal{U}(\mathcal{P})} t_0 \to^m_{\mathcal{L}/\mathcal{U}(\mathcal{P})} t_m$. Then there exists a derivation D' which has the form $t'_0 \to^m_{\mathcal{L}/\mathcal{U}(\mathcal{P})} t'_m \to^*_{\mathcal{U}(\mathcal{P})} s'_0 \to^n_{\mathcal{K}/\mathcal{U}(\mathcal{P})} s'_n$ with $t'_0 \in \mathcal{T}^\sharp_\mathsf{c}$.*

Proof. Consider the following two dependency pairs: $1\colon u^\sharp_k \to \mathrm{COM}(v^\sharp_{k1}, \ldots, v^\sharp_{kr})$ and $2\colon u^\sharp_l \to \mathrm{COM}(v^\sharp_{l1}, \ldots, u^\sharp_{lr})$. Here the dependency pair 1 belongs to $\mathcal{K}$ and denotes the last dependency pair employed in D before the path leaves $\mathcal{K}$ into $\mathcal{L}$, while 2 denotes the first pair in $\mathcal{L}$. The assumption that there is no edge connecting $\mathcal{K}$ and $\mathcal{L}$ can be reformulated as follows:

(†) No context C and no substitutions $\sigma, \tau\colon \mathcal{V} \to \mathcal{T}(\mathcal{F}, \mathcal{V})$ exist such that $\mathrm{COM}(v^\sharp_{k1}\sigma, \ldots, v^\sharp_{kr}\sigma) \to^*_{\mathcal{U}(\mathcal{P})} C[u^\sharp_l\tau]$ holds.

To prove the lemma, we proceed by induction on n. It suffices to consider the step case $n > 1$. By assumption the last rewrite step in the subderivation $D_0\colon s_0 \to^n_{\mathcal{K}/\mathcal{U}(\mathcal{P})} s_n$ employs dependency pair 1. Let $p \in \mathcal{P}\mathsf{os}(s_n)$ denote the position of the reduct $\mathrm{COM}(v^\sharp_{k1}\tau, \ldots, v^\sharp_{kr}\tau)$ in s_n. By assumption there exists a derivation $s_n \to^*_{\mathcal{U}(\mathcal{P})} t_0$. Let $q \in \mathcal{P}\mathsf{os}(s_n)$ denote the position of the redex in s_n that is contracted as first step in this reduction. Without loss of generality we can assume that both positions are parallel to each other. Otherwise one of the following cases applies. Either $p < q$ or $p \geqslant q$. But clearly the first case contradicts the assumption (†). Hence, assume the second. But this is also impossible. Lemma 20 yields that $s_n|_q \in \mathcal{T}^\sharp_\mathsf{c}$, which contradicts that q is redex with respect to $\mathcal{U}(\mathcal{P})$. Repeating this argument we see that position p has exactly one descendant in t_0. A similar argument shows that all redex positions in the subderivation $D_1\colon t_0 \to^m_{\mathcal{L}/\mathcal{U}(\mathcal{P})} t_m$ are parallel to (descendants of) p. Hence, we can move the last rewrite step $s_{n-1} \to_\mathcal{K} s_n$ in the derivation D_0 after the derivation D_1. Note that in each of the terms $(t_i)_{i=1,\ldots,m}$ the position p exists and denotes the term $\mathrm{COM}(v^\sharp_{k1}\tau, \ldots, v^\sharp_{kr}\tau)$. Hence, the replacement of $\mathrm{COM}(v^\sharp_{k1}\tau, \ldots, v^\sharp_{kr}\tau)$ everywhere by $u^\sharp_k\sigma$ does not affect the validity of the rewrite sequence. Furthermore the set $\mathcal{T}^\sharp_\mathsf{c}$ is closed under this operation. Now, induction hypothesis becomes applicable to derive the existence of the sought derivation D'. □

Let $\mathcal{G}$ denote a weak or weak innermost dependency graph and let $D\colon s \to^{\ell} t$ denote a derivation, such that $s \in \mathcal{T}^{\sharp}_{\mathsf{b}}$. Here $\to$ denotes either $\to_{\mathcal{P}/\mathcal{U}(\mathcal{P})}$ or $\xrightarrow{\mathrm{i}}_{\mathcal{P}/\mathcal{U}(\mathcal{P})}$. We say that D is *based on* $(\mathcal{P}_1, \dots, \mathcal{P}_m)$ in $\mathcal{G}/_{\equiv}$ if D is of form

$$s \xrightarrow{(\mathrm{i})}{}^{\ell_1}_{\mathcal{P}_1/\mathcal{U}(\mathcal{P})} \cdots \xrightarrow{(\mathrm{i})}{}^{\ell_m}_{\mathcal{P}_m/\mathcal{U}(\mathcal{P})} t\ ,$$

with $\ell_1, \dots, \ell_m \geqslant 0$. We arrive at the main lemma of this section.

Lemma 22. *Let $\mathcal{P}$ denote a set of weak or weak innermost dependency pairs, let $s \in \mathcal{T}^{\sharp}_{\mathsf{b}}$ and let $D\colon s \to^{\ell} t$ denote a maximal derivation, where $\to$ denotes $\to_{\mathcal{P}/\mathcal{U}(\mathcal{P})}$ or $\xrightarrow{\mathrm{i}}_{\mathcal{P}/\mathcal{U}(\mathcal{P})}$ respectively. Suppose that D is based on $(\mathcal{P}_1, \dots, \mathcal{P}_m)$ and $\mathcal{P}_1 \in \mathsf{Src}$. Then there exists a derivation $D'\colon s \to^{\ell} t$ based on $(\mathcal{P}'_1, \dots, \mathcal{P}'_{m'})$, with $\mathcal{P}'_1 \in \mathsf{Src}$ such that all $\mathcal{P}'_i$ $(i \in \{1, \dots, m'\})$ are pairwise distinct.*

Proof. Without loss of generality, we restrict our attention to weak dependency pairs. To prove the lemma, we consider a sequence $(\mathcal{P}_1, \dots, \mathcal{P}_m)$, where there exist indices i, j and k with $i < j < k$ and $\mathcal{P}_i = \mathcal{P}_k$. By induction on $j - i$ we show that this path is transformable into a sequence $(\mathcal{P}'_1, \dots, \mathcal{P}'_{m'})$ of the required form. It suffices to prove the step case. Moreover, we can assume without loss of generality that $k = j + 1$. Consider the two dependency pairs: $1\colon l^{\sharp}_j \to \mathrm{COM}(u^{\sharp}_{j1}, \dots, u^{\sharp}_{jr})$ and $2\colon l^{\sharp}_k \to \mathrm{COM}(u^{\sharp}_{k1}, \dots, u^{\sharp}_{kr})$. Dependency pair 1 belongs to $\mathcal{P}_j$ and denotes the last dependency pair employed in D before the sequence leaves $\mathcal{P}_j$ into $\mathcal{P}_k$, while 2 denotes the first pair in $\mathcal{P}_k$. We consider two cases:

1. Assume there exist a context C and substitutions $\sigma, \tau\colon \mathcal{V} \to \mathcal{T}(\mathcal{F}, \mathcal{V})$ such that the following holds: $\mathrm{COM}(u^{\sharp}_{j1}\sigma, \dots, u^{\sharp}_{jr}\sigma) \to^{*} C[l^{\sharp}_k\tau]$. Thus by definition of weak dependency graphs the node in $\mathsf{WDG}(\mathcal{R})$ representing dependency pair 1 is connected to the node representing dependency pair 2. In particular every node in the SCCs represented by $\mathcal{P}_i = \mathcal{P}_k$ is connected to every node in the SCC represented by $\mathcal{P}_j$. This implies that $\mathcal{P}_i = \mathcal{P}_j = \mathcal{P}_k$ contradicting the assumption.
2. Otherwise, there is no edge between $\mathcal{P}_j$ and $\mathcal{P}_k$ in the graph $\mathcal{G}/_{\equiv}$ and by the assumptions on $(\mathcal{P}_1, \dots, \mathcal{P}_\ell)$ we find a derivation of the following form: $D_0\colon s_{j_1} \to^{p}_{\mathcal{P}_j/\mathcal{U}(\mathcal{P})} s_{j_p} \to^{*}_{\mathcal{U}(\mathcal{P})} s_{k_1} \to^{q}_{\mathcal{P}_k/\mathcal{U}(\mathcal{P})} s_{k_q}$. Due to Lemma 21 there exists a derivation $D_1\colon s'_{k_1} \to^{q}_{\mathcal{P}_k/\mathcal{U}(\mathcal{P})} s'_{k_q} \to^{*}_{\mathcal{U}(\mathcal{P})} s'_{j_1} \to^{p}_{\mathcal{P}_j/\mathcal{U}(\mathcal{P})} s'_{j_p}$ so that the number of (weak) dependency pair steps is unchanged. The sequence $(\mathcal{P}_1, \dots, \mathcal{P}_j, \mathcal{P}_k, \dots, \mathcal{P}_m)$ is reorderable into $(\mathcal{P}_1, \dots, \mathcal{P}_k, \mathcal{P}_j, \dots, \mathcal{P}_m)$ without affecting the length ℓ of the $\to_{\mathcal{P}/\mathcal{U}(\mathcal{P})}$-rewrite sequence. By assumption $k = j + 1$, hence induction hypothesis becomes applicable and we conclude the existence of a path $(\mathcal{P}'_1, \dots, \mathcal{P}'_{m'})$ fulfilling the assertions of the lemma. □

Finally, we arrive at the main contribution of this paper.

Theorem 23. *Let $\mathcal{R}$ be a TRS, let $\mathcal{P}$ be the set of weak or weak innermost dependency pairs, let A denotes the maximum arity of compound symbols and let K denotes the number of SCCs in the weak (innermost) dependency graph $\mathcal{G}$. Suppose $t \in \mathcal{T}^{\sharp}_{\mathsf{b}}$ is (innermost) terminating and define*

$L(t) := \max\{\mathsf{dl}(t, \to_{\mathcal{P}_m/\mathcal{S}}) \mid (\mathcal{P}_1, \dots, \mathcal{P}_m)$ *is a path in* $\mathcal{G}/_{\equiv}$ *such that* $\mathcal{P}_1 \in \mathsf{Src}\}$, *where* $\mathcal{S} = \mathcal{P}_1 \cup \dots \cup \mathcal{P}_{m-1} \cup \mathcal{U}(\mathcal{P}_1 \cup \dots \cup \mathcal{P}_m)$. *Then* $\mathsf{dl}(t, \xrightarrow{(i)}_{\mathcal{P}/\mathcal{U}(\mathcal{P})}) \leqslant A^K \cdot K \cdot L(t)$.

Proof. Let $(\mathcal{P}_1, \dots, \mathcal{P}_m)$ be a path in $\mathcal{P}/_{\equiv}$ such that $\mathcal{P}_1 \in \mathsf{Src}$ and let $D\colon t \to^{\ell} u$, denote a maximal derivation based on this path. (Here $\to$ denotes $\to_{\mathcal{P}/\mathcal{U}(\mathcal{P})}$ or $\xrightarrow{i}_{\mathcal{P}/\mathcal{U}(\mathcal{P})}$.) Lemma 22 yields that D has the following form:

$$t = t_0 \to^{\ell_1}_{\mathcal{P}_1/\mathcal{U}(\mathcal{P}_1)} t_{\ell_1} \to^{\ell_2}_{\mathcal{P}_2/\mathcal{U}(\mathcal{P}_1)\cup\mathcal{U}(\mathcal{P}_2)} \cdots \to^{\ell_m}_{\mathcal{P}_m/\mathcal{U}(\mathcal{P}_1)\cup\dots\cup\mathcal{U}(\mathcal{P}_m)} t_n = u \;, \quad (4)$$

where $t_0 \in \mathcal{T}^{\sharp}_{\mathsf{b}}$ and $t_i \in \mathcal{T}^{\sharp}_{\mathsf{c}}$ for all $i \geqslant 1$. It suffices to estimate ℓ_j for all $j = 1, \dots, m$ suitably. Let j be arbitrary, but fixed. Consider the subderivation D' of (4) where m is replaced by j. Clearly D' is contained in the following derivation:

$$t \to^{*}_{\mathcal{P}_1\cup\dots\cup\mathcal{P}_{i-1}\cup\mathcal{U}(\mathcal{P}_1)\cup\dots\cup\mathcal{U}(\mathcal{P}_{j-1})} \cdot \to^{\ell_j}_{\mathcal{P}_j/\mathcal{U}(\mathcal{P}_1)\cup\dots\cup\mathcal{U}(\mathcal{P}_j)} t_{\ell_j}$$

Hence Lemma 17 is applicable, thus $\ell_j \leqslant \mathsf{dl}(t, \to_{\mathcal{P}_j/\mathcal{P}_1\cup\dots\cup\mathcal{P}_{j-1}\cup\mathcal{U}(\mathcal{P}_1)\cup\dots\cup\mathcal{U}(\mathcal{P}_j)})$. As $\mathcal{U}(\mathcal{P}_1) \cup \dots \cup \mathcal{U}(\mathcal{P}_j) \subseteq \mathcal{U}(\mathcal{P}_1 \cup \dots \cup \mathcal{P}_j)$ we conclude $\ell_j \leqslant L(t)$ and obtain $\ell = \ell_1 + \ell_2 + \dots + \ell_m \leqslant K \cdot L(t)$.

Above we argued that any connected component in $\mathcal{P}/_{\equiv}$ is a tree. Clearly the number of nodes in this tree is less than $\frac{A^K-1}{A-1}$ and an arbitrary derivation can at most be based on $\frac{A^K-1}{A-1}$-many different paths. As the length of a derivation D based on a specific path can be estimated by $K \cdot L(t)$, we conclude that the length of an arbitrary derivation is less than $\frac{A^K-1}{A-1} \cdot K \cdot L(t) \leqslant A^K \cdot K \cdot L(t)$. This completes the proof of the theorem. □

Theorem 23 together with Proposition 9 form a suitable analog of Theorem 13: Let $\mathcal{P}$ be the set of weak or weak innermost dependency pairs. Suppose for every path $(\mathcal{P}_1, \dots, \mathcal{P}_m)$ in $\mathcal{P}$ there exist an SLI $\mathcal{A}_m$ compatible with the usable rules of $\bigcup_{i=1}^{m} \mathcal{P}_i$. Assume the existence of a safe and G-collapsible reduction pairs $(\gtrsim_m, \succ_m)$ such that $\mathcal{U}(\bigcup_{i=1}^{m} \mathcal{P}_i) \cup \bigcup_{i=1}^{m-1} \mathcal{P}_i$ is compatible with $\gtrsim_m$ and $\mathcal{P}_m$ compatible with $\succ_m$. Then for any $t \in \mathcal{T}_{\mathsf{b}}$ the derivation height $\mathsf{dl}(t, \xrightarrow{(i)})$ with respect to (innermost) rewriting is linearly bounded in $\mathsf{G}(t^{\sharp}, \succ_m)$ and $|t|$.

Corollary 24. *Let* $\mathcal{R}$ *be a TRS, let* $\mathcal{P}$ *be the set of weak (innermost) dependency pairs, and let* $\mathcal{G}$ *denote the weak (innermost) dependency graph. Suppose for every path* $(\mathcal{P}_1, \dots, \mathcal{P}_m)$ *in* $\mathcal{G}/_{\equiv}$ *there exist an SLI* $\mathcal{A}_m$ *and linear (quadratic) restricted interpretations* $\mathcal{B}_m$ *such that* $(\geqslant_{\mathcal{B}_m}, >_{\mathcal{B}_m})$ *forms a safe reduction pair with (i)* $\mathcal{U}(\mathcal{P}_1 \cup \dots \cup \mathcal{P}_m) \subseteq >_{\mathcal{A}_m}$ *(ii)* $\mathcal{P}_1 \cup \dots \cup \mathcal{P}_{m-1} \cup \mathcal{U}(\mathcal{P}_1 \cup \dots \cup \mathcal{P}_m) \subseteq \geqslant_{\mathcal{B}_m}$, *and (iii)* $\mathcal{P}_m \subseteq >_{\mathcal{B}_m}$. *Then the runtime complexity of a TRS* $\mathcal{R}$ *is linear or quadratic, respectively.*

Proof. Observe that the assumptions imply that any basic term $t \in \mathcal{T}_{\mathsf{b}}$ is terminating with respect to $\mathcal{R}$: Any infinite derivation with respect to $\mathcal{R}$ starting in t can be translated into an infinite derivation with respect to $\mathcal{U}(\mathcal{R}) \cup \mathcal{P}$ (see [1, Lemma 16]). Moreover, as the number of paths in $\mathcal{G}/_{\equiv}$ is finite, there exists a component $\mathcal{P}_i$ that represents an infinite rewrite sequence. This is a contradiction. Without loss of generality, we assume $\mathcal{P} = \mathsf{WDP}(\mathcal{P})$ and $\mathcal{G} = \mathsf{WDG}(\mathcal{P})$. Notice that the reduction pair $(\geqslant_{\mathcal{B}_m}, >_{\mathcal{B}_m})$ is safe and collapsible. Hence for

all m, the length of any $\to_{\mathcal{P}_m/\mathcal{S}}$-rewrite sequence is less than $p_m(|t|)$, where p_m denotes a linear (or quadratic) polynomial, depending on $|t|$ only. (Here $\mathcal{S} = \mathcal{P}_1 \cup \cdots \cup \mathcal{P}_{m-1} \cup \mathcal{U}(\mathcal{P}_1 \cup \cdots \cup \mathcal{P}_m)$.) In analogy to the operator L, we define $M(t) := \max\{\mathsf{dl}(t, \to_{\mathcal{P}_m \cup \mathcal{S}}) \mid (\mathcal{P}_1, \ldots, \mathcal{P}_m) \text{ is a path in } \mathcal{G}/_{\equiv} \text{ such that } \mathcal{P}_1 \in \mathsf{Src}\}$. An application of Proposition 9 yields $M(t) = \mathsf{O}(p_m(|t|))$. Following the pattern of the proof of the Theorem, we establish the existence of a polynomial p such that $\mathsf{dl}(t, \to_{\mathcal{P} \cup \mathcal{U}(\mathcal{P})}) \leqslant p(|t|)$ holds for any basic term t. Finally, the corollary follows by an application of Proposition 8. □

As mentioned above, in the dependency graph refinement for termination analysis it suffices to guarantee for each cycle $\mathcal{C}$ that there exists no $\mathcal{C}$-minimal rewrite sequences. For that one only needs to find a reduction pair $(\gtrsim, \succ)$ such that $\mathcal{R} \subseteq \gtrsim$, $\mathcal{C} \subseteq \gtrsim$ and $\mathcal{C} \cap \succ \neq \varnothing$. Thus, considering Theorem 23 it is tempting to think that it should suffice to replace strongly connected components by cycles and the stronger conditions should apply. However this intuition is deceiving as shown by the next example.

Example 25. Consider the TRS $\mathcal{R}$ of $\mathsf{f}(\mathsf{s}(x), \mathsf{0}) \to \mathsf{f}(x, \mathsf{s}(x))$ and $\mathsf{f}(x, \mathsf{s}(y)) \to \mathsf{f}(x, y)$. $\mathsf{WDP}(\mathcal{R})$ consists of $1\colon \mathsf{f}^\sharp(\mathsf{s}(x), \mathsf{0}) \to \mathsf{f}^\sharp(x, \mathsf{s}(x))$ and $2\colon \mathsf{f}^\sharp(x, \mathsf{s}(y)) \to \mathsf{f}^\sharp(x, y)$, and the weak dependency graph $\mathsf{WDG}(\mathcal{R})$ contains two cycles $\{1, 2\}$ and $\{2\}$. There are two linear restricted interpretations $\mathcal{A}$ and $\mathcal{B}$ such that $\{1, 2\} \subseteq \geqslant_{\mathcal{A}} \cup >_{\mathcal{A}}$, $\{1\} \subseteq >_{\mathcal{A}}$, and $\{1\} \subseteq >_{\mathcal{B}}$. Here, however, we must not conclude linear runtime complexity, because the runtime complexity of $\mathcal{R}$ is at least quadratic.

5 Conclusion

In this section we provide (experimental) evidence on the applicability of the technique for complexity analysis established in this paper. We briefly consider the efficient implementation of the techniques provided by Theorem 23 and Corollary 24. Firstly, in order to approximate (weak) dependency graphs, we adapted (innermost) dependency graph estimations using the functions TCAP (ICAP) [14]. Secondly, note that a graph including n nodes may contain an exponential number of paths. However, to apply Corollary 24 it is sufficient to handle only paths in the following set. Notice that this set contains at most n^2 paths.

$$\{(\mathcal{P}_1, \ldots, \mathcal{P}_k) \mid (\mathcal{P}_1, \ldots, \mathcal{P}_m) \text{ is a maximal path and } k \leqslant m\}\ ,$$

Example 26 (continued from Example 16). For $\mathsf{WDG}(\mathcal{R})/_{\equiv}$ the above set consists of 8 paths: $(\{13\})$, $(\{13\}, \{11\})$, $(\{13\}, \{12\})$, $(\{15\})$, $(\{15\}, \{14\})$, $(\{17\})$, $(\{18, 19, 20\})$, and $(\{18, 19, 20\}, \{16\})$. In the following we only consider the last three paths, since all other paths are similarly handled.

- Consider $(\{17\})$. Note $\mathcal{U}(\{17\}) = \varnothing$. By taking an arbitrary SLI $\mathcal{A}$ and the linear restricted interpretation $\mathcal{B}$ with $\mathsf{gcd}^\sharp_{\mathcal{B}}(x, y) = x$ and $\mathsf{s}_{\mathcal{B}}(x) = x + 1$, we have $\varnothing \subseteq >_{\mathcal{A}}$, $\varnothing \subseteq \geqslant_{\mathcal{B}}$, and $\{17\} \subseteq >_{\mathcal{B}}$.
- Consider $(\{18, 19, 20\})$. Note $\mathcal{U}(\{18, 19, 20\}) = \{1, \ldots, 5\}$. By taking the SLI $\mathcal{A}$ and the linear restricted interpretation $\mathcal{B}$ with $\mathsf{0}_{\mathcal{A}} = \mathsf{true}_{\mathcal{A}} = \mathsf{false}_{\mathcal{A}} = 0$, $\mathsf{s}_{\mathcal{A}}(x) = x+1$, $x -_{\mathcal{A}} y = x \leqslant_{\mathcal{A}} y = x+y+1$; $\mathsf{0}_{\mathcal{B}} = \mathsf{true}_{\mathcal{B}} = \mathsf{false}_{\mathcal{B}} = x \leqslant_{\mathcal{B}} y = 0$,

Table 1. Results for Linear Runtime Complexities

		full rewriting			*innermost rewriting*		
	direct	Prop.8	Prop.10	Cor.24	Prop.8	Prop.10	Cor.24
success	139	138	119	137	143	128	147
			(161)	(179)		(170)	(189)
	15	*21*	*18*	*52*	*21*	*21*	*65*
failure	1535	1518	1560	1510	1511	1551	1499
	1789	*3185*	*152*	*1690*	*3214*	*214*	*1625*
timeout	5	23	0	32	25	0	33

Table 2. Results for Quadratic Runtime Complexities

		full rewriting			*innermost rewriting*		
	direct	Prop.8	Prop.10	Cor.24	Prop.8	Prop.10	Cor.24
success	179	172	125	141	172	126	146
			(191)	(210)		(192)	(213)
	623	*732*	*295*	*616*	*725*	*278*	*787*
failure	745	699	1499	1434	699	1496	1431
	4431	*4522*	*1128*	*2883*	*4536*	*1062*	*2856*
timeout	753	807	55	104	807	57	102

$\mathsf{s}_{\mathcal{B}}(x) = x + 2$, $x -_{\mathcal{B}} y = x$, $\mathsf{gcd}^{\sharp}_{\mathcal{B}}(x, y) = x + y + 1$, and $\mathsf{if}_{\mathsf{gcd}}{}^{\sharp}_{\mathcal{B}}(x, y, z) = y + z$, we obtain $\{1, \ldots, 5\} \subseteq >_{\mathcal{A}}$, $\{1, \ldots, 5\} \subseteq \geqslant_{\mathcal{B}}$, and $\{18, 19, 20\} \subseteq >_{\mathcal{B}}$.

– Consider $(\{18, 19, 20\}, \{16\})$. Note $\mathcal{U}(\{16\}) = \varnothing$. By taking the same $\mathcal{A}$ and also $\mathcal{B}$, we have $\{1, \ldots, 5\} \subseteq >_{\mathcal{A}}$, $\{1, \ldots, 5, 18, 19, 20\} \subseteq \geqslant_{\mathcal{B}}$, and $\{16\} \subseteq >_{\mathcal{B}}$.

Thus, all path constraints are handled by linear restricted interpretations. Hence, the runtime complexity function of $\mathcal{R}$ is linear.

Moreover, to deal efficiently with polynomial interpretations, the issuing constraints are encoded in *propositional logic* in a similar spirit as in [16]. Assignments are found by employing a state-of-the-art SAT solver, in our case MiniSat[5]. Furthermore, SLIs are handled by linear programming. Based on these ideas we implemented a complexity analyser. As suitable test bed we used the rewrite systems in the Termination Problem Data Base version 4.0.[6] The presented tests were performed single-threaded on a 1.50 GHz Intel® Core™ Duo Processor L2300 and 1.5 GB of memory. For each system we used a timeout of 60 seconds. In interpreting defined and dependency pair symbols, we restrict the search to polynomials in the range $\{0, 1, \ldots, 5\}$. Table 1 (2) shows the experimental results for linear (quadratic) runtime complexities based on linear (quadratic) restricted interpretations.[7] Text written in *italics* below the number of successes or failures indicates total time (in seconds) of success cases or failure cases, respectively.[8]

[5] http://minisat.se/.

[6] See http://www.lri.fr/~marche/tpdb/.

[7] For full experimental evidence see http://www.jaist.ac.jp/~hirokawa/08b/.

[8] Sum of numbers in each column may be less than 1679 because of stack overflow.

The columns marked "Prop. 10" and "Cor. 24" refer to the applicability of the respective results. For sake of comparison, in the parentheses we indicate the number of successes by the method of the column *or* by Proposition 8.

In concluding, we observe that the experimental data shows that the here introduced dependency graph refinement for complexity analysis extends the analytic power of the methods introduced in [1]. Notice the significant difference between those TRSs that can be handled by Propositions 8, 10 in contrast to those that can be handled either by Proposition 8 or by Corollary 24. Moreover observe the gain in power in relation to direct methods, compare also [3,4].

References

1. Hirokawa, N., Moser, G.: Automated complexity analysis based on the dependency pair method. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS, vol. 5195, pp. 364–379. Springer, Heidelberg (2008)
2. Lescanne, P.: Termination of rewrite systems by elementary interpretations. Formal Aspects of Computing 7(1), 77–90 (1995)
3. Bonfante, G., Cichon, A., Marion, J.Y., Touzet, H.: Algorithms with polynomial interpretation termination proof. JFP 11(1), 33–53 (2001)
4. Avanzini, M., Moser, G.: Complexity analysis by rewriting. In: Garrigue, J., Hermenegildo, M.V. (eds.) FLOPS 2008. LNCS, vol. 4989, pp. 130–146. Springer, Heidelberg (2008)
5. Arts, T., Giesl, J.: Termination of term rewriting using dependency pairs. TCS 236, 133–178 (2000)
6. Arts, T., Giesl, J.: A collection of examples for termination of term rewriting using dependency pairs. Technical Report AIB-2001-09, RWTH Aachen (2001)
7. Giesl, J., Arts, T., Ohlebusch, E.: Modular termination proofs for rewriting using dependency pairs. JSC 34(1), 21–58 (2002)
8. Hirokawa, N., Middeldorp, A.: Tyrolean termination tool: Techniques and features. IC 205, 474–511 (2007)
9. Giesl, J., Thiemann, R., Schneider-Kamp, P., Falke, S.: Mechanizing and improving dependency pairs. JAR 37(3), 155–203 (2006)
10. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1998)
11. Terese: Term Rewriting Systems. Cambridge Tracts in Theoretical Computer Science, vol. 55. Cambridge University Press, Cambridge (2003)
12. Hofbauer, D., Lautemann, C.: Termination proofs and the length of derivations. In: Dershowitz, N. (ed.) RTA 1989. LNCS, vol. 355, pp. 167–177. Springer, Heidelberg (1989)
13. Contejean, E., Marché, C., Tomás, A.P., Urbain, X.: Mechanically proving termination using polynomial interpretations. JAR 34(4), 325–363 (2005)
14. Giesl, J., Thiemann, R., Schneider-Kamp, P.: Proving and disproving termination of higher-order functions. In: Gramlich, B. (ed.) FroCos 2005. LNCS (LNAI), vol. 3717, pp. 216–231. Springer, Heidelberg (2005)
15. Hirokawa, N., Middeldorp, A.: Automating the dependency pair method. IC 199(1,2), 172–199 (2005)
16. Fuhs, C., Giesl, J., Middeldorp, A., Schneider-Kamp, P., Thiemann, R., Zankl, H.: SAT solving for termination analysis with polynomial interpretations. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 340–354. Springer, Heidelberg (2007)

Uncurrying for Termination*

Nao Hirokawa[1], Aart Middeldorp[2], and Harald Zankl[2]

[1] School of Information Science
Japan Advanced Institute of Science and Technology, Japan
hirokawa@jaist.ac.jp
[2] Institute of Computer Science
University of Innsbruck, Austria
{aart.middeldorp,harald.zankl}@uibk.ac.at

Abstract. First-order applicative term rewrite systems provide a natural framework for modeling higher-order aspects. In this paper we present a transformation from untyped applicative term rewrite systems to functional term rewrite systems that preserves and reflects termination. Our transformation is less restrictive than other approaches. In particular, head variables in right-hand sides of rewrite rules can be handled. To further increase the applicability of our transformation, we present a version for dependency pairs.

1 Introduction

In this paper we are concerned with proving termination of first-order applicative term rewrite systems. These systems provide a natural framework for modeling higher-order aspects found in functional programming languages. The signature of an applicative term rewrite system consists of constants and a single binary function symbol called application and denoted by the infix and left-associative symbol $\circ$. Proving termination of applicative term rewrite systems is challenging because the rewrite rules lack sufficient structure. As a consequence, simplification orders are not effective as $\circ$ is the only function symbol of non-zero arity. Moreover, the dependency pair method is of little help as $\circ$ is the only defined non-constant symbol.

The main contribution of this paper is a new transformation that recovers the structure in applicative rewrite rules. Our transformation can deal with partial applications as well as head variables in right-hand sides of rewrite rules. The key ingredient is the addition of sufficiently many uncurrying rules to the transformed system. These rules are also crucial for a smooth transition into the dependency pair framework. Unlike the transformation of applicative dependency pair problems presented in [10,17], our uncurrying processor preserves minimality (cf. Section 6), which means that it can be used at any node in a modular (non-)termination proof attempt.

* This research is supported by FWF (Austrian Science Fund) project P18763, Grant-in-Aid for Young Scientists 20800022 of the Ministry of Education, Culture, Sports, Science and Technology of Japan, and STARC.

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 667–681, 2008.

The remainder of this paper is organised as follows. After recalling existing results in Section 2, we present a new uncurrying transformation and prove that it is sound and complete for termination in Section 3. Despite its simplicity, the transformation has some subtleties which are illustrated by several examples. Two extensions to the dependency pair framework are presented in Section 4. Our results are empirically evaluated in Section 5 and we conclude with a discussion of related work in Section 6. Parts of Section 3 were first announced in a note by the first two authors that was presented at the 3rd International Workshop on Higher-Order Rewriting (Seattle, 2006).

2 Preliminaries

We assume familiarity with term rewriting [5] in general and termination [20] in particular.

Definition 1. *An* applicative *term rewrite system (ATRS for short) is a TRS over a signature that consists of constants and a single binary function symbol called application denoted by the infix and left-associative symbol* $\circ$*. In examples we often use juxtaposition instead of* $\circ$*.*

Every ordinary TRS can be transformed into an applicative rewrite system by currying.

Definition 2. *Let* $\mathcal{F}$ *be a signature. The currying system* $\mathcal{C}(\mathcal{F})$ *consists of the rewrite rules* $f_{i+1}(x_1,\dots,x_i,y) \to f_i(x_1,\dots,x_i) \circ y$ *for every* n*-ary function symbol* $f \in \mathcal{F}$ *and every* $0 \leqslant i < n$*. Here* $f_n = f$ *and, for every* $0 \leqslant i < n$*,* f_i *is a fresh function symbol of arity* i*.*

The currying system $\mathcal{C}(\mathcal{F})$ is confluent and terminating. Hence every term t has a unique normal form $t{\downarrow_{\mathcal{C}(\mathcal{F})}}$.

Definition 3. *Let* $\mathcal{R}$ *be a TRS over the signature* $\mathcal{F}$*. The* curried *system* $\mathcal{R}{\downarrow_{\mathcal{C}(\mathcal{F})}}$ *is the ATRS consisting of the rules* $l{\downarrow_{\mathcal{C}(\mathcal{F})}} \to r{\downarrow_{\mathcal{C}(\mathcal{F})}}$ *for every* $l \to r \in \mathcal{R}$*. The signature of* $\mathcal{R}{\downarrow_{\mathcal{C}(\mathcal{F})}}$ *contains the application symbol* $\circ$ *and a constant* f_0 *for every function symbol* $f \in \mathcal{F}$*.*

In the following we write $\mathcal{R}{\downarrow_{\mathcal{C}}}$ for $\mathcal{R}{\downarrow_{\mathcal{C}(\mathcal{F})}}$ whenever $\mathcal{F}$ can be inferred from the context or is irrelevant. Moreover, we write f for f_0.

Example 4. The TRS $\mathcal{R} = \{\mathsf{0}{+}y \to y, \mathsf{s}(x){+}y \to \mathsf{s}(x{+}y)\}$ is transformed into the ATRS $\mathcal{R}{\downarrow_{\mathcal{C}}} = \{+\ \mathsf{0}\ y \to y, +\ (\mathsf{s}\ x)\ y \to \mathsf{s}\ (+\ x\ y)\}$. Every rewrite sequence in $\mathcal{R}$ can be transformed into a sequence in $\mathcal{R}{\downarrow_{\mathcal{C}}}$, but the reverse does not hold. For instance, with respect to the above example, the rewrite step $+\ (\mathsf{s}\ (+\ \mathsf{0}))\ \mathsf{0} \to \mathsf{s}\ (+\ (+\ \mathsf{0})\ \mathsf{0})$ in $\mathcal{R}{\downarrow_{\mathcal{C}}}$ does not correspond to a rewrite step in $\mathcal{R}$. Nevertheless, termination of $\mathcal{R}$ implies termination of $\mathcal{R}{\downarrow_{\mathcal{C}}}$.

Theorem 5 (Kennaway *et al.* [15])**.** *A TRS* $\mathcal{R}$ *is terminating if and only if* $\mathcal{R}{\downarrow_{\mathcal{C}}}$ *is terminating.* □

As an immediate consequence we get the following transformation method for proving termination of ATRSs.

Corollary 6. *An ATRS $\mathcal{R}$ is terminating if and only if there exists a terminating TRS $\mathcal{S}$ such that $\mathcal{S}{\downarrow_{\mathcal{C}}} = \mathcal{R}$ (modulo renaming).* □

In [10] this method is called *transformation $\mathcal{A}$*. As can be seen from the following example, the method does not handle partially applied terms and, more seriously, head variables. Hence the method is of limited applicability as it cannot cope with the higher-order aspects modeled by ATRSs.

Example 7. Consider the ATRS $\mathcal{R}$ (from [2])

$$\begin{array}{lll} 1\colon & \mathsf{id}\ x \to x & 4\colon \quad \mathsf{map}\ f\ \mathsf{nil} \to \mathsf{nil} \\ 2\colon & \mathsf{add}\ 0 \to \mathsf{id} & 5\colon \mathsf{map}\ f\ (:\ x\ y) \to :\ (f\ x)\ (\mathsf{map}\ f\ y) \\ 3\colon & \mathsf{add}\ (\mathsf{s}\ x)\ y \to \mathsf{s}\ (\mathsf{add}\ x\ y) & \end{array}$$

Rules 1 and 4 are readily translated into functional form: $\mathsf{id}_1(x) \to x$ and $\mathsf{map}_2(f, \mathsf{nil}) \to \mathsf{nil}$. However, we cannot find functional forms for rules 2 and 3 because the 'arity' of add is 1 in rule 2 and 2 in rule 3. Because of the presence of the head variable f in the subterm $f\ x$, there is no functional term t such that $t{\downarrow_{\mathcal{C}}} = :\ (f\ x)\ (\mathsf{map}\ f\ y)$. Hence also rule 5 cannot be transformed.

3 Uncurrying

In this section we present an uncurrying transformation that can deal with ATRSs like in Example 7. Throughout this section we assume that $\mathcal{R}$ is an ATRS over a signature $\mathcal{F}$.

Definition 8. *The* applicative arity $\mathrm{aa}(f)$ *of a constant $f \in \mathcal{F}$ is defined as the maximum n such that $f \circ t_1 \circ \cdots \circ t_n$ is a subterm in the left- or right-hand side of a rule in $\mathcal{R}$. This notion is extended to terms as follows:*

$$\mathrm{aa}(t) = \begin{cases} \mathrm{aa}(f) & \text{if } t \text{ is a constant } f \\ \mathrm{aa}(t_1) - 1 & \text{if } t = t_1 \circ t_2 \end{cases}$$

Note that $\mathrm{aa}(t)$ *is undefined if the head symbol of t is a variable.*

Definition 9. *The uncurrying system $\mathcal{U}(\mathcal{F})$ consists of the following rewrite rules $f_i(x_1, \ldots, x_i) \circ y \to f_{i+1}(x_1, \ldots, x_i, y)$ for every constant $f \in \mathcal{F}$ and every $0 \leqslant i < \mathrm{aa}(f)$. Here $f_0 = f$ and, for every $i > 0$, f_i is a fresh function symbol of arity i. We say that $\mathcal{R}$ is* left head variable free *if* $\mathrm{aa}(t)$ *is defined for every non-variable subterm t of a left-hand side of a rule in $\mathcal{R}$. This means that no subterm of a left-hand side in $\mathcal{R}$ is of the form $t_1 \circ t_2$ where t_1 is a variable.*

The uncurrying system $\mathcal{U}(\mathcal{F})$, or simply $\mathcal{U}$, is confluent and terminating. Hence every term t has a unique normal form $t{\downarrow_{\mathcal{U}}}$.

Definition 10. *The* uncurried *system* $\mathcal{R}{\downarrow_{\mathcal{U}}}$ *is the TRS consisting of the rules* $l{\downarrow_{\mathcal{U}}} \to r{\downarrow_{\mathcal{U}}}$ *for every* $l \to r \in \mathcal{R}$.

Example 11. The ATRS $\mathcal{R}$ of Example 7 is transformed into $\mathcal{R}{\downarrow_{\mathcal{U}}}$:

$$\begin{aligned} \mathsf{id}_1(x) &\to x & \mathsf{map}_2(f, \mathsf{nil}) &\to \mathsf{nil} \\ \mathsf{add}_1(0) &\to \mathsf{id} & \mathsf{map}_2(f, {:}_2(x, y)) &\to {:}_2(f \circ x, \mathsf{map}_2(f, y)) \\ \mathsf{add}_2(\mathsf{s}_1(x), y) &\to \mathsf{s}_1(\mathsf{add}_2(x, y)) \end{aligned}$$

The TRS $\mathcal{R}{\downarrow_{\mathcal{U}}}$ is an obvious candidate for $\mathcal{S}$ in Corollary 6. However, as can be seen from the following example, the rules of $\mathcal{R}{\downarrow_{\mathcal{U}}}$ are not enough to simulate an arbitrary rewrite sequence in $\mathcal{R}$.

Example 12. The non-terminating ATRS $\mathcal{R} = \{\mathsf{id}\ x \to x, \mathsf{f}\ x \to \mathsf{id}\ \mathsf{f}\ x\}$ is transformed into the terminating TRS $\mathcal{R}{\downarrow_{\mathcal{U}}} = \{\mathsf{id}_1(x) \to x, \mathsf{f}_1(x) \to \mathsf{id}_2(\mathsf{f}, x)\}$. Note that $\mathcal{R}{\downarrow_{\mathcal{U}}}{\downarrow_{\mathcal{C}}} = \{\mathsf{id}_1\ x \to x, \mathsf{f}_1\ x \to \mathsf{id}_2\ \mathsf{f}\ x\}$ is different from $\mathcal{R}$.

In the above example we need rules that connect id_2 and id_1 as well as f_1 and f. The natural idea is now to add $\mathcal{U}(\mathcal{F})$. In the following we write $\mathcal{U}^+(\mathcal{R}, \mathcal{F})$ for $\mathcal{R}{\downarrow_{\mathcal{U}(\mathcal{F})}} \cup \mathcal{U}(\mathcal{F})$. If $\mathcal{F}$ can be inferred from the context or is irrelevant, $\mathcal{U}^+(\mathcal{R}, \mathcal{F})$ is abbreviated to $\mathcal{U}^+(\mathcal{R})$.

Example 13. Consider the ATRS $\mathcal{R}$ in Example 12. We have $\mathrm{aa}(\mathsf{id}) = 2$ and $\mathrm{aa}(\mathsf{f}) = 1$. The TRS $\mathcal{U}^+(\mathcal{R})$ consists of the following rules

$$\begin{aligned} \mathsf{id}_1(x) &\to x & \mathsf{id} \circ x &\to \mathsf{id}_1(x) & \mathsf{f} \circ x &\to \mathsf{f}_1(x) \\ \mathsf{f}_1(x) &\to \mathsf{id}_2(\mathsf{f}, x) & \mathsf{id}_1(x) \circ y &\to \mathsf{id}_2(x, y) \end{aligned}$$

and is easily shown to be terminating.

As the above example shows, we do not yet have a sound transformation. The ATRS $\mathcal{R}$ admits the cycle $\mathsf{f}\ x \to \mathsf{id}\ \mathsf{f}\ x \to \mathsf{f}\ x$. In $\mathcal{U}^+(\mathcal{R})$ we have $\mathsf{f}_1(x) \to \mathsf{id}_2(\mathsf{f}, x)$ but the term $\mathsf{id}_2(\mathsf{f}, x)$ does not rewrite to $\mathsf{f}_1(x)$. It would if the rule $\mathsf{id}\ x\ y \to x\ y$ were present in $\mathcal{R}$. This inspires the following definition.

Definition 14. *Let* $\mathcal{R}$ *be a left head variable free ATRS. The* η-saturated *ATRS* $\mathcal{R}_\eta$ *is the smallest extension of* $\mathcal{R}$ *such that* $l \circ x \to r \circ x \in \mathcal{R}_\eta$ *whenever* $l \to r \in \mathcal{R}_\eta$ *and* $\mathrm{aa}(l) > 0$. *Here* x *is a variable that does not appear in* $l \to r$.

The rules added during η-saturation do not affect the termination behaviour of $\mathcal{R}$, according to the following lemma whose straightforward proof is omitted. Moreover, $\mathcal{R}_\eta$ is left head variable free if and only if $\mathcal{R}$ is left head variable free.

Lemma 15. *If* $\mathcal{R}$ *is a left head variable free ATRS then* $\to_{\mathcal{R}} = \to_{\mathcal{R}_\eta}$. □

We can now state the main result of this section.

Theorem 16. *A left head variable free ATRS* $\mathcal{R}$ *is terminating if* $\mathcal{U}^+(\mathcal{R}_\eta)$ *is terminating.*

It is important to note that the applicative arities used in the definition of $\mathcal{U}^+(\mathcal{R}_\eta)$ are computed before η-saturation.

Example 17. The non-terminating ATRS $\mathcal{R} = \{\mathsf{f} \to \mathsf{g}\ \mathsf{a}, \mathsf{g} \to \mathsf{f}\}$ is transformed into $\mathcal{U}^+(\mathcal{R}_\eta) = \{\mathsf{f} \to \mathsf{g}_1(\mathsf{a}), \mathsf{g} \to \mathsf{f}, \mathsf{g}_1(x) \to \mathsf{f} \circ x, \mathsf{g} \circ x \to \mathsf{g}_1(x)\}$ because $\mathrm{aa}(\mathsf{f}) = 0$. The resulting TRS is non-terminating. Uncurrying with $\mathrm{aa}(\mathsf{f}) = 1$ produces the terminating TRS $\{\mathsf{f} \to \mathsf{g}_1(\mathsf{a}), \mathsf{g} \to \mathsf{f}, \mathsf{g}_1(x) \to \mathsf{f}_1(x), \mathsf{g} \circ x \to \mathsf{g}_1(x), \mathsf{f} \circ x \to \mathsf{f}_1(x)\}$.

Before presenting the proof of Theorem 16, we revisit Example 7.

Example 18. Consider again the ATRS $\mathcal{R}$ of Example 7. Proving termination of the transformed TRS $\mathcal{U}^+(\mathcal{R}_\eta)$

$$
\begin{array}{rcl}
\mathsf{id}_1(x) \to x & : \circ x \to :_1(x) & \mathsf{id} \circ x \to \mathsf{id}_1(x) \\
\mathsf{add}_1(0) \to \mathsf{id} & :_1(x) \circ y \to :_2(x, y) & \mathsf{add} \circ x \to \mathsf{add}_1(x) \\
\mathsf{add}_2(0, y) \to \mathsf{id}_1(y) & & \mathsf{add}_1(x) \circ y \to \mathsf{add}_2(x, y) \\
\mathsf{add}_2(\mathsf{s}_1(x), y) \to \mathsf{s}_1(\mathsf{add}_2(x, y)) & & \mathsf{s} \circ x \to \mathsf{s}_1(x) \\
\mathsf{map}_2(f, \mathsf{nil}) \to \mathsf{nil} & & \mathsf{map} \circ x \to \mathsf{map}_1(x) \\
\mathsf{map}_2(f, :_2(x, y)) \to :_2(f \circ x, \mathsf{map}_2(f, y)) & & \mathsf{map}_1(x) \circ y \to \mathsf{map}_2(x, y)
\end{array}
$$

is straightforward with the dependency pair method (recursive SCC algorithm with three applications of the subterm criterion).

The following two lemmata state factorisation properties which are used in the proof of Theorem 16. The easy induction proofs are omitted.

Lemma 19. *Let s and t be terms. If $\mathrm{aa}(s) > 0$ then $s{\downarrow_\mathcal{U}} \circ t{\downarrow_\mathcal{U}} \to_\mathcal{U}^* (s \circ t){\downarrow_\mathcal{U}}$. If $\mathrm{aa}(s) \leqslant 0$ or if $\mathrm{aa}(s)$ is undefined then $s{\downarrow_\mathcal{U}} \circ t{\downarrow_\mathcal{U}} = (s \circ t){\downarrow_\mathcal{U}}$.* □

For a substitution σ, we write $\sigma{\downarrow_\mathcal{U}}$ for the substitution $\{x \mapsto \sigma(x){\downarrow_\mathcal{U}} \mid x \in \mathcal{V}\}$.

Lemma 20. *Let σ be a substitution. For every term t, $t{\downarrow_\mathcal{U}}\sigma{\downarrow_\mathcal{U}} \to_\mathcal{U}^* (t\sigma){\downarrow_\mathcal{U}}$. If t is head variable free then $t{\downarrow_\mathcal{U}}\sigma{\downarrow_\mathcal{U}} = (t\sigma){\downarrow_\mathcal{U}}$.* □

Proof (of Theorem 16). We show that $s{\downarrow_\mathcal{U}} \to^+_{\mathcal{U}^+(\mathcal{R}_\eta)} t{\downarrow_\mathcal{U}}$ whenever $s \to_{\mathcal{R}_\eta} t$. This entails that any infinite $\mathcal{R}_\eta$ derivation is transformed into an infinite $\mathcal{U}^+(\mathcal{R}_\eta)$ derivation. The theorem follows from this observation and Lemma 15. Let $s = C[l\sigma]$ and $t = C[r\sigma]$ with $l \to r \in \mathcal{R}_\eta$. We use induction on the size of the context C.

- If $C = \Box$ then $s{\downarrow_\mathcal{U}} = (l\sigma){\downarrow_\mathcal{U}} = l{\downarrow_\mathcal{U}}\sigma{\downarrow_\mathcal{U}}$ and $r{\downarrow_\mathcal{U}}\sigma{\downarrow_\mathcal{U}} \to_\mathcal{U}^* (r\sigma){\downarrow_\mathcal{U}} = t{\downarrow_\mathcal{U}}$ by Lemma 20. Hence $s{\downarrow_\mathcal{U}} \to^+_{\mathcal{U}^+(\mathcal{R}_\eta)} t{\downarrow_\mathcal{U}}$.
- Suppose $C = \Box \circ s_1 \circ \cdots \circ s_n$ and $n > 0$. Since $\mathcal{R}_\eta$ is left head variable free, $\mathrm{aa}(l)$ is defined. If $\mathrm{aa}(l) = 0$ then

$$
\begin{aligned}
s{\downarrow_\mathcal{U}} &= (l\sigma \circ s_1 \circ \cdots \circ s_n){\downarrow_\mathcal{U}} = l\sigma{\downarrow_\mathcal{U}} \circ s_1{\downarrow_\mathcal{U}} \circ \cdots \circ s_n{\downarrow_\mathcal{U}} \\
&= l{\downarrow_\mathcal{U}}\sigma{\downarrow_\mathcal{U}} \circ s_1{\downarrow_\mathcal{U}} \circ \cdots \circ s_n{\downarrow_\mathcal{U}}
\end{aligned}
$$

  and

$$\begin{aligned} r{\downarrow_{\mathcal{U}}}\sigma{\downarrow_{\mathcal{U}}} \circ s_1{\downarrow_{\mathcal{U}}} \circ \cdots \circ s_n{\downarrow_{\mathcal{U}}} &\to_{\mathcal{U}}^* (r\sigma){\downarrow_{\mathcal{U}}} \circ s_1{\downarrow_{\mathcal{U}}} \circ \cdots \circ s_n{\downarrow_{\mathcal{U}}} \\ &\to_{\mathcal{U}}^* (r\sigma \circ s_1 \circ \cdots \circ s_n){\downarrow_{\mathcal{U}}} = t{\downarrow_{\mathcal{U}}} \end{aligned}$$

by applications of Lemmata 19 and 20. Hence $s{\downarrow_{\mathcal{U}}} \to_{\mathcal{U}^+(\mathcal{R}_\eta)}^+ t{\downarrow_{\mathcal{U}}}$. If $\mathrm{aa}(l) > 0$ then $l \circ x \to r \circ x \in \mathcal{R}_\eta$ for some fresh variable x. We have $s = C'[(l \circ x)\tau]$ and $t = C'[(r \circ x)\tau]$ for the context $C' = \Box \circ s_2 \circ \cdots \circ s_n$ and the substitution $\tau = \sigma \cup \{x \mapsto s_1\}$. Since C' is smaller than C, we can apply the induction hypothesis which yields the desired result.

– In the remaining case $C = s_1 \circ C'$. The induction hypothesis yields

$$C'[l\sigma]{\downarrow_{\mathcal{U}}} \to_{\mathcal{U}^+(\mathcal{R}_\eta)}^+ C'[r\sigma]{\downarrow_{\mathcal{U}}}$$

If $\mathrm{aa}(s_1) \leqslant 0$ or if $\mathrm{aa}(s_1)$ is undefined then $s{\downarrow_{\mathcal{U}}} = s_1{\downarrow_{\mathcal{U}}} \circ C'[l\sigma]{\downarrow_{\mathcal{U}}}$ and $t{\downarrow_{\mathcal{U}}} = s_1{\downarrow_{\mathcal{U}}} \circ C'[r\sigma]{\downarrow_{\mathcal{U}}}$ by Lemma 19. If $\mathrm{aa}(s_1) > 0$ then $s_1{\downarrow_{\mathcal{U}}} = f_i(u_1, \ldots, u_i)$ for the head symbol f of s_1 and some terms $u_1, \ldots, u_i$. So

$$s{\downarrow_{\mathcal{U}}} = f_{i+1}(u_1, \ldots, u_i, C'[l\sigma]{\downarrow_{\mathcal{U}}})$$

and

$$t{\downarrow_{\mathcal{U}}} = f_{i+1}(u_1, \ldots, u_i, C'[r\sigma]{\downarrow_{\mathcal{U}}})$$

Hence in both cases we obtain $s{\downarrow_{\mathcal{U}}} \to_{\mathcal{U}^+(\mathcal{R}_\eta)}^+ t{\downarrow_{\mathcal{U}}}$. □

The next example shows that the left head variable freeness condition cannot be weakened to the well-definedness of $\mathrm{aa}(l)$ for every left-hand side l.

Example 21. Consider the non-terminating ATRS $\mathcal{R} = \{\mathsf{f}\ (x\ \mathsf{a}) \to \mathsf{f}\ (\mathsf{g}\ \mathsf{b}), \mathsf{g}\ \mathsf{b} \to \mathsf{h}\ \mathsf{a}\}$. The transformed TRS $\mathcal{U}^+(\mathcal{R}_\eta)$ consists of the rules

$$\begin{array}{lll} \mathsf{f}_1(x \circ \mathsf{a}) \to \mathsf{f}_1(\mathsf{g}_1(\mathsf{b})) & \mathsf{f} \circ x \to \mathsf{f}_1(x) & \mathsf{h} \circ x \to \mathsf{h}_1(x) \\ \mathsf{g}_1(\mathsf{b}) \to \mathsf{h}_1(\mathsf{a}) & \mathsf{g} \circ x \to \mathsf{g}_1(x) & \end{array}$$

and is terminating because its rules are oriented from left to right by the lexicographic path order with precedence $\circ \succ \mathsf{g}_1 \succ \mathsf{f}_1 \succ \mathsf{h}_1 \succ \mathsf{a} \succ \mathsf{b}$. Note that $\mathrm{aa}(\mathsf{f}\ (x\ \mathsf{a})) = 0$.

The uncurrying transformation is not always useful.

Example 22. Consider the one-rule TRS $\mathcal{R} = \{\mathsf{C}\ x\ y\ z\ u \to x\ z\ (x\ y\ z\ u)\}$ from [7]. The termination of $\mathcal{R}$ is proved by the lexicographic path order with empty precedence. The transformed TRS $\mathcal{U}^+(\mathcal{R}_\eta)$ consists of

$$\begin{array}{ll} \mathsf{C}_4(x, y, z, u) \to x \circ z \circ (x \circ y \circ z \circ u) & \\ \mathsf{C} \circ x \to \mathsf{C}_1(x) & \mathsf{C}_2(x, y) \circ z \to \mathsf{C}_3(x, y, z) \\ \mathsf{C}_1(x) \circ y \to \mathsf{C}_2(x, y) & \mathsf{C}_3(x, y, z) \circ u \to \mathsf{C}_4(x, y, z, u) \end{array}$$

None of the tools that participated in the termination competitions between 2005 and 2007 is able to prove the termination of this TRS.

We show that the converse of Theorem 16 also holds. Hence the uncurrying transformation is not only sound but also complete for termination. (This does not contradict the preceding example.)

Definition 23. *For a term t over the signature of the TRS $\mathcal{U}^+(\mathcal{R})$, we denote by $t{\downarrow_{\mathcal{C}'}}$ the result of identifying different function symbols in $t{\downarrow_{\mathcal{C}}}$ that originate from the same function symbol in $\mathcal{F}$. The notation ${\downarrow_{\mathcal{C}'}}$ is extended to TRSs and substitutions in the obvious way.*

Example 24. For the ATRS $\mathcal{R}$ of Example 12 we have $\mathcal{R}{\downarrow_{\mathcal{U}}}{\downarrow_{\mathcal{C}'}} = \mathcal{R}$.

Lemma 25. *For every t, C, and σ, $C[t\sigma]{\downarrow_{\mathcal{C}'}} = C{\downarrow_{\mathcal{C}'}}[t{\downarrow_{\mathcal{C}'}}\sigma{\downarrow_{\mathcal{C}'}}]$.*

Proof. Straightforward induction on C and t. □

Lemma 26. *Let $\mathcal{R}$ be a left head variable free ATRS. If s and t are terms over the signature of $\mathcal{U}^+(\mathcal{R})$ then $s \to_{\mathcal{R}_\eta{\downarrow_{\mathcal{U}}}} t$ if and only if $s{\downarrow_{\mathcal{C}'}} \to_{\mathcal{R}_\eta} t{\downarrow_{\mathcal{C}'}}$.*

Proof. This follows from Lemma 25 and the fact that $\mathcal{R}_\eta{\downarrow_{\mathcal{U}}}{\downarrow_{\mathcal{C}'}} = \mathcal{R}_\eta$. □

Lemma 27. *Let $\mathcal{R}$ be a left head variable free ATRS. If s and t are terms over the signature of $\mathcal{U}^+(\mathcal{R})$ and $s \to_{\mathcal{U}} t$ then $s{\downarrow_{\mathcal{C}'}} = t{\downarrow_{\mathcal{C}'}}$.*

Proof. This follows from Lemma 25 in connection with the observation that all rules in $\mathcal{U}{\downarrow_{\mathcal{C}'}}$ have equal left- and right-hand sides. □

Theorem 28. *If a left head variable free ATRS $\mathcal{R}$ is terminating then $\mathcal{U}^+(\mathcal{R}_\eta)$ is terminating.*

Proof. Assume that $\mathcal{U}^+(\mathcal{R}_\eta)$ is non-terminating. Since $\mathcal{U}$ is terminating, any infinite rewrite sequence has the form $s_1 \to_{\mathcal{R}_\eta{\downarrow_{\mathcal{U}}}} t_1 \to_{\mathcal{U}}^* s_2 \to_{\mathcal{R}_\eta{\downarrow_{\mathcal{U}}}} t_2 \to_{\mathcal{U}}^* \cdots$. Applications of Lemmata 26 and 27 transform this sequence into $s_1{\downarrow_{\mathcal{C}'}} \to_{\mathcal{R}_\eta} t_1{\downarrow_{\mathcal{C}'}} = s_2{\downarrow_{\mathcal{C}'}} \to_{\mathcal{R}_\eta} t_2{\downarrow_{\mathcal{C}'}} = \cdots$. It follows that $\mathcal{R}_\eta$ is non-terminating. Since $\to_{\mathcal{R}} = \to_{\mathcal{R}_\eta}$ by Lemma 15, we conclude that $\mathcal{R}$ is non-terminating. □

We conclude this section by describing a trivial mirroring technique for TRSs. This technique can be used to eliminate some of the left head variables in an ATRS.

Definition 29. *Let t be a term. The term t^M is defined as follows: $t^M = t$ if t is a variable and $t^M = f(t_n^M, \ldots, t_1^M)$ if $t = f(t_1, \ldots, t_n)$. Moreover, if $\mathcal{R}$ is a TRS then $\mathcal{R}^M = \{l^M \to r^M \mid l \to r \in \mathcal{R}\}$.*

We obviously have $s \to_{\mathcal{R}} t$ if and only if $s^M \to_{\mathcal{R}^M} t^M$. This gives the following result.

Theorem 30. *A TRS $\mathcal{R}$ is terminating if and only if $\mathcal{R}^M$ is terminating.* □

Example 31. Consider the one-rule ATRS $\mathcal{R} = \{x\ (\mathsf{a}\ \mathsf{a})\ \mathsf{a} \to \mathsf{a}\ (\mathsf{a}\ \mathsf{a})\ x\}$. While $\mathcal{R}$ has a head variable in its left-hand side, the mirrored version $\mathcal{R}^M = \{\mathsf{a}\ (\mathsf{a}\ \mathsf{a})\ x \to x\ (\mathsf{a}\ \mathsf{a}\ \mathsf{a})\}$ is left head variable free. The transformed TRS $\mathcal{U}^+((\mathcal{R}^M)_\eta)$

$$\mathsf{a}_2(\mathsf{a}_1(\mathsf{a}), x) \to x \circ \mathsf{a}_2(\mathsf{a}, \mathsf{a}) \qquad \mathsf{a} \circ x \to \mathsf{a}_1(x) \qquad \mathsf{a}_1(x) \circ y \to \mathsf{a}_2(x, y)$$

is easily proved terminating with dependency pairs and a linear polynomial interpretation.

4 Uncurrying with Dependency Pairs

In this section we incorporate the uncurrying transformation into the dependency pair framework [4,9,11,13,17]. Let $\mathcal{R}$ be a TRS over a signature $\mathcal{F}$. The signature $\mathcal{F}$ is extended with *dependency pair symbols* $f^\sharp$ for every symbol $f \in \{\mathrm{root}(l) \mid l \to r \in \mathcal{R}\}$, where $f^\sharp$ has the same arity as f, resulting in the signature $\mathcal{F}^\sharp$. If $l \to r \in \mathcal{R}$ and t is a subterm of r with a defined root symbol that is not a proper subterm of l then the rule $l^\sharp \to t^\sharp$ is a *dependency pair* of $\mathcal{R}$. Here $l^\sharp$ and $t^\sharp$ are the result of replacing the root symbols in l and t by the corresponding dependency pair symbols. The set of dependency pairs of $\mathcal{R}$ is denoted by $\mathsf{DP}(\mathcal{R})$. A *DP problem* is a pair of TRSs $(\mathcal{P}, \mathcal{R})$ such that the root symbols of the rules in $\mathcal{P}$ do neither occur in $\mathcal{R}$ nor in proper subterms of the left- and right-hand sides of rules in $\mathcal{P}$. The problem is said to be *finite* if there is no infinite sequence $s_1 \xrightarrow{\epsilon}_{\mathcal{P}} t_1 \to^*_{\mathcal{R}} s_2 \xrightarrow{\epsilon}_{\mathcal{P}} t_2 \to^*_{\mathcal{R}} \cdots$ such that all terms $t_1, t_2, \ldots$ are terminating with respect to $\mathcal{R}$. Such an infinite sequence is said to be *minimal*. The main result underlying the dependency pair approach states that termination of a TRS $\mathcal{R}$ is equivalent to finiteness of the DP problem $(\mathsf{DP}(\mathcal{R}), \mathcal{R})$.

In order to prove a DP problem finite, a number of *DP processors* have been developed. DP processors are functions that take a DP problem as input and return a set of DP problems as output. In order to be employed to prove termination they need to be *sound*, that is, if all DP problems in a set returned by a DP processor are finite then the initial DP problem is finite. In addition, to ensure that a DP processor can be used to prove non-termination it must be *complete* which means that if one of the DP problems returned by the DP processor is not finite then the original DP problem is not finite.

In this section we present two DP processors that uncurry applicative DP problems, which are DP problems over applicative signatures containing two application symbols: $\circ$ and $\circ^\sharp$.

4.1 Uncurrying Processor

Definition 32. *Let $(\mathcal{P}, \mathcal{R})$ be an applicative DP problem. The DP processor $\mathcal{U}_1$ is defined as*

$$(\mathcal{P}, \mathcal{R}) \mapsto \begin{cases} \{(\mathcal{P}{\downarrow_{\mathcal{U}(\mathcal{F})}}, \mathcal{U}^+(\mathcal{R}_\eta, \mathcal{F}))\} & \text{if } \mathcal{P} \cup \mathcal{R} \text{ is left head variable free} \\ \{(\mathcal{P}, \mathcal{R})\} & \text{otherwise} \end{cases}$$

where $\mathcal{F}$ consists of all function symbols of $\mathcal{P} \cup \mathcal{R}$ minus the root symbols of $\mathcal{P}$.

Theorem 33. *The DP processor $\mathcal{U}_1$ is sound and complete.*

Proof. Let $\mathcal{F}$ be the set of function symbols of $\mathcal{P} \cup \mathcal{R}$ minus the root symbols of $\mathcal{P}$. We first show soundness. Let $(\mathcal{P}, \mathcal{R})$ be an applicative DP problem with the property that $\mathcal{P} \cup \mathcal{R}$ is left head variable free. Suppose the DP problem $(\mathcal{P}{\downarrow_{\mathcal{U}}}, \mathcal{U}^+(\mathcal{R}_\eta))$ is finite. We have to show that $(\mathcal{P}, \mathcal{R})$ is finite. Suppose to the contrary that $(\mathcal{P}, \mathcal{R})$ is not finite. So there exists a minimal rewrite sequence

$$s_1 \xrightarrow{\epsilon}_{\mathcal{P}} t_1 \to^*_{\mathcal{R}} s_2 \xrightarrow{\epsilon}_{\mathcal{P}} t_2 \to^*_{\mathcal{R}} \cdots \tag{1}$$

By Lemmata 15 and 20 together with the claim in the proof of Theorem 16, this sequence can be transformed into $s_1{\downarrow_{\mathcal{U}}} \xrightarrow{\epsilon}_{\mathcal{P}\downarrow_{\mathcal{U}}} u_1 \to^*_{\mathcal{U}} t_1{\downarrow_{\mathcal{U}}} \to^*_{\mathcal{U}^+(\mathcal{R}_\eta)} s_2{\downarrow_{\mathcal{U}}} \xrightarrow{\epsilon}_{\mathcal{P}\downarrow_{\mathcal{U}}} u_2 \to^*_{\mathcal{U}} t_2{\downarrow_{\mathcal{U}}} \to^*_{\mathcal{U}^+(\mathcal{R}_\eta)} \cdots$. It remains to show that all terms $u_1, u_2, \ldots$ are terminating with respect to $\mathcal{U}^+(\mathcal{R}_\eta)$. Fix i. We have $u_i{\downarrow_{\mathcal{C}'}} = t_i{\downarrow_{\mathcal{U}}}{\downarrow_{\mathcal{C}'}} = t_i$. Due to the minimality of (1), t_i is terminating with respect to $\mathcal{R}$ and, according to Lemma 15, also with respect to $\mathcal{R}_\eta$. Hence, due to the proof of Theorem 28, u_i is terminating with respect to $\mathcal{U}^+(\mathcal{R}_\eta)$.

Next we show completeness of the DP processor $\mathcal{U}_1$. So let $(\mathcal{P}, \mathcal{R})$ be an applicative DP problem with the property that $\mathcal{P} \cup \mathcal{R}$ is left head variable free and suppose that the DP problem $(\mathcal{P}{\downarrow_{\mathcal{U}}}, \mathcal{U}^+(\mathcal{R}_\eta))$ is not finite. So there exists a minimal rewrite sequence $s_1 \xrightarrow{\epsilon}_{\mathcal{P}\downarrow_{\mathcal{U}}} t_1 \to^*_{\mathcal{U}^+(\mathcal{R}_\eta)} s_2 \xrightarrow{\epsilon}_{\mathcal{P}\downarrow_{\mathcal{U}}} t_2 \to^*_{\mathcal{U}^+(\mathcal{R}_\eta)} \cdots$. Using Lemmata 26 and 27 this sequence can be transformed into $s_1{\downarrow_{\mathcal{C}'}} \xrightarrow{\epsilon}_{\mathcal{P}} t_1{\downarrow_{\mathcal{C}'}} \to^*_{\mathcal{R}_\eta} s_2{\downarrow_{\mathcal{C}'}} \xrightarrow{\epsilon}_{\mathcal{P}} t_2{\downarrow_{\mathcal{C}'}} \to^*_{\mathcal{R}_\eta} \cdots$. In order to conclude that the DP problem $(\mathcal{P}, \mathcal{R})$ is not finite, it remains to show that the terms $t_1{\downarrow_{\mathcal{C}'}}, t_2{\downarrow_{\mathcal{C}'}}, \ldots$ are terminating with respect to $\mathcal{R}_\eta$. This follows from the assumption that the terms $t_1, t_2, \ldots$ are terminating with respect to $\mathcal{U}^+(\mathcal{R}_\eta)$ in connection with Lemma 26. □

The following example from [17] shows that the $\mathcal{A}$ transformation of [10] is not sound because it does not preserve minimality.[1]

Example 34. Consider the applicative DP problem $(\mathcal{P}, \mathcal{R})$ with $\mathcal{P}$ consisting of the rewrite rule $(\mathsf{g}\ x)\ (\mathsf{h}\ y)^\sharp\ z \to z\ z^\sharp\ z$ and $\mathcal{R}$ consisting of the rules

$$\mathsf{c}\ x\ y \to x \qquad \mathsf{c}\ (\mathsf{g}\ x)\ y \to \mathsf{c}\ (\mathsf{g}\ x)\ y$$
$$\mathsf{c}\ x\ y \to y \qquad \mathsf{c}\ x\ (\mathsf{g}\ y) \to \mathsf{c}\ x\ (\mathsf{g}\ y)$$

The DP problem $(\mathcal{P}, \mathcal{R})$ is not finite because of the following minimal rewrite sequence:

$$\begin{aligned}(\mathsf{g}\ x)\ (\mathsf{h}\ x)^\sharp\ (\mathsf{c}\ \mathsf{g}\ \mathsf{h}\ x) &\xrightarrow{\epsilon}_{\mathcal{P}} (\mathsf{c}\ \mathsf{g}\ \mathsf{h}\ x)\ (\mathsf{c}\ \mathsf{g}\ \mathsf{h}\ x)^\sharp\ (\mathsf{c}\ \mathsf{g}\ \mathsf{h}\ x)\\ &\to_{\mathcal{R}} (\mathsf{g}\ x)\ (\mathsf{c}\ \mathsf{g}\ \mathsf{h}\ x)^\sharp\ (\mathsf{c}\ \mathsf{g}\ \mathsf{h}\ x)\\ &\to_{\mathcal{R}} (\mathsf{g}\ x)\ (\mathsf{h}\ x)^\sharp\ (\mathsf{c}\ \mathsf{g}\ \mathsf{h}\ x)\end{aligned}$$

Applying the DP processor $\mathcal{U}_1$ produces $(\mathcal{P}{\downarrow_{\mathcal{U}}}, \mathcal{U}^+(\mathcal{R}_\eta))$ with $\mathcal{P}{\downarrow_{\mathcal{U}}}$ consisting of the rewrite rule $\mathsf{g}_1(x) \circ \mathsf{h}_1(y) \circ^\sharp z \to z \circ z \circ^\sharp z$ and $\mathcal{U}^+(\mathcal{R}_\eta)$ consisting of the rules

$$\begin{array}{lll}\mathsf{c}_2(x,y) \to x & \mathsf{c}_2(\mathsf{g}_1(x),y) \to \mathsf{c}_2(\mathsf{g}_1(x),y) & \mathsf{g} \circ x \to \mathsf{g}_1(x)\\ \mathsf{c}_2(x,y) \to y & \mathsf{c}_2(x,\mathsf{g}_1(y)) \to \mathsf{c}_2(x,\mathsf{g}_1(y)) & \mathsf{h} \circ x \to \mathsf{h}_1(x)\\ \mathsf{c} \circ x \to \mathsf{c}_1(x) & \mathsf{c}_1(x) \circ y \to \mathsf{c}_2(x,y) & \end{array}$$

This DP problem is not finite:

$$\begin{aligned}\mathsf{g}_1(x) \circ \mathsf{h}_1(x) \circ^\sharp (\mathsf{c}_2(\mathsf{g},\mathsf{h}) \circ x) &\xrightarrow{\epsilon}_{\mathcal{P}\downarrow_{\mathcal{U}}} (\mathsf{c}_2(\mathsf{g},\mathsf{h}) \circ x) \circ (\mathsf{c}_2(\mathsf{g},\mathsf{h}) \circ x) \circ^\sharp (\mathsf{c}_2(\mathsf{g},\mathsf{h}) \circ x)\\ &\to^*_{\mathcal{U}^+(\mathcal{R}_\eta)} (\mathsf{g} \circ x) \circ (\mathsf{h} \circ x) \circ^\sharp (\mathsf{c}_2(\mathsf{g},\mathsf{h}) \circ x)\\ &\to^*_{\mathcal{U}^+(\mathcal{R}_\eta)} \mathsf{g}_1(x) \circ \mathsf{h}_1(x) \circ^\sharp (\mathsf{c}_2(\mathsf{g},\mathsf{h}) \circ x)\end{aligned}$$

Note that $\mathsf{c}_2(\mathsf{g},\mathsf{h}) \circ x$ is terminating with respect to $\mathcal{U}^+(\mathcal{R}_\eta)$.

[1] Since minimality is not part of the definition of finite DP problems in [10], this does not contradict the results in [10].

The uncurrying rules are essential in this example, even though in the original DP problem all occurrences of each constant have the same number of arguments. Indeed, the $\mathcal{A}$ transformation leaves out the uncurrying rules, resulting in a DP problem that admits infinite rewrite sequences but no minimal ones since one has to instantiate the variable z in $\mathsf{g}_1(x) \circ \mathsf{h}_1(y) \circ^\sharp z \to z \circ z \circ^\sharp z$ by a term that contains a subterm of the form $\mathsf{c}_2(\mathsf{g}_1(s), t)$ or $\mathsf{c}_2(s, \mathsf{g}_1(t))$ and the rules $\mathsf{c}_2(\mathsf{g}_1(x), y) \to \mathsf{c}_2(\mathsf{g}_1(x), y)$ and $\mathsf{c}_2(x, \mathsf{g}_1(y)) \to \mathsf{c}_2(x, \mathsf{g}_1(y))$ ensure that these terms are non-terminating.

4.2 Freezing

A drawback of $\mathcal{U}_1$ is that dependency pair symbols are excluded from the uncurrying process. Typically, all pairs in $\mathcal{P}$ have the same root symbol $\circ^\sharp$. The next example shows that uncurrying root symbols of $\mathcal{P}$ can be beneficial.

Example 35. After processing the ATRS consisting of the rule $\mathsf{a}\ x\ \mathsf{a} \to \mathsf{a}\ (\mathsf{a}\ \mathsf{a})\ x$ with the recursive SCC algorithm and $\mathcal{U}_1$, the rule $\mathsf{a}_1(x) \circ^\sharp \mathsf{a} \to \mathsf{a}_1(\mathsf{a}_1(\mathsf{a})) \circ^\sharp x$ must be oriented. This cannot be done with a linear polynomial interpretation. If we transform the rule into $\mathsf{a}_2^\sharp(x, \mathsf{a}) \to \mathsf{a}_2^\sharp(\mathsf{a}_1(\mathsf{a}), x)$ this becomes trivial.

To this end we introduce a simple variant of *freezing* [19].

Definition 36. *A* simple freeze *is a partial mapping $\Bbbk$ that assigns to a function symbol of arity $n > 0$ an argument position $i \in \{1, \dots, n\}$. Every simple freeze $\Bbbk$ induces the following partial mapping on non-variable terms $t = f(t_1, \dots, t_n)$, also denoted by $\Bbbk$:*

- *if $\Bbbk(f)$ is undefined or $n = 0$ then $\Bbbk(t) = t$,*
- *if $\Bbbk(f) = i$ and $t_i = g(u_1, \dots, u_m)$ then*

$$\Bbbk(t) = \Bbbk_g^f(t_1, \dots, t_{i-1}, u_1, \dots, u_m, t_{i+1}, \dots, t_n)$$

  where $\Bbbk_g^f$ is a fresh $m + n - 1$-ary function symbol,
- *if $\Bbbk(f) = i$ and t_i is a variable then $\Bbbk(t)$ is undefined.*

We denote $\{\Bbbk(l) \to \Bbbk(r) \mid l \to r \in \mathcal{R}\}$ by $\Bbbk(\mathcal{R})$.

Now uncurrying for dependency pair symbols is formulated with the simple freeze $\Bbbk(\circ^\sharp) = 1$, transforming $f_n(t_1, \dots, t_n) \circ^\sharp t_{n+1}$ to $\Bbbk_{f_n}^{\circ^\sharp}(t_1, \dots, t_n, t_{n+1})$. Writing $f_{n+1}^\sharp$ for $\Bbbk_{f_n}^{\circ^\sharp}$, we obtain the uncurried term $f_{n+1}^\sharp(t_1, \dots, t_n, t_{n+1})$. In Example 35 we have $\Bbbk(\{\mathsf{a}_1(x) \circ^\sharp \mathsf{a} \to \mathsf{a}_1(\mathsf{a}_1(\mathsf{a})) \circ^\sharp x\}) = \{\mathsf{a}_2^\sharp(x, \mathsf{a}) \to \mathsf{a}_2^\sharp(\mathsf{a}_1(\mathsf{a}), x)\}$.

Definition 37. *A term t is* strongly root stable *with respect to a TRS $\mathcal{R}$ if $t\sigma \to_\mathcal{R}^* \cdot \xrightarrow{\epsilon}_\mathcal{R} u$ does not hold for any substitution σ and term u. Let $\Bbbk$ be a simple freeze. A DP problem $(\mathcal{P}, \mathcal{R})$ is* $\Bbbk$-stable *if $\Bbbk(\mathcal{P})$ is well-defined and t_i is strongly root stable for $\mathcal{R}$ whenever $s \to f(t_1, \dots, t_n) \in \mathcal{P}$ and $\Bbbk(f) = i$.*

Definition 38. *Let $(\mathcal{P},\mathcal{R})$ be a DP problem and $\circledast$ a simple freeze. The DP processor $\circledast$ is defined as*

$$(\mathcal{P},\mathcal{R}) \mapsto \begin{cases} \{(\circledast(\mathcal{P}),\mathcal{R})\} & \text{if } (\mathcal{P},\mathcal{R}) \text{ is } \circledast\text{-stable} \\ \{(\mathcal{P},\mathcal{R})\} & \text{otherwise} \end{cases}$$

Furthermore, the DP processor $\mathcal{U}_2$ is defined as the composition $\circledast \circ \mathcal{U}_1$, where $\circledast(\circ^\sharp) = 1$.

Theorem 39. *The DP processor $\circledast$ is sound and complete.*

Proof. We show that every minimal rewrite sequence $s_1 \xrightarrow{\epsilon}_{\mathcal{P}} t_1 \to^*_{\mathcal{R}} s_2 \xrightarrow{\epsilon}_{\mathcal{P}} t_2 \to^*_{\mathcal{R}} \cdots$ can be transformed into the minimal sequence $\circledast(s_1) \xrightarrow{\epsilon}_{\circledast(\mathcal{P})} \circledast(t_1) \to^*_{\mathcal{R}} \circledast(s_2) \xrightarrow{\epsilon}_{\circledast(\mathcal{P})} \circledast(t_2) \to^*_{\mathcal{R}} \cdots$ and vice versa. This follows from the following three observations.

- $s_i \xrightarrow{\epsilon}_{\mathcal{P}} t_i$ if and only if $\circledast(s_i) \xrightarrow{\epsilon}_{\circledast(\mathcal{P})} \circledast(t_i)$

  We have $s_i \xrightarrow{\epsilon}_{\mathcal{P}} t_i$ if and only if $s_i = l\sigma$ and $t_i = r\sigma$ with $l \to r \in \mathcal{P}$. Since $\circledast(\mathcal{P})$ is well-defined, the latter is equivalent to $\circledast(s_i) = \circledast(l\sigma) = \circledast(l)\sigma \xrightarrow{\epsilon}_{\circledast(\mathcal{P})} \circledast(r)\sigma = \circledast(r\sigma) = \circledast(t_i)$.

- $t_i \to^*_{\mathcal{R}} s_{i+1}$ if and only if $\circledast(t_i) \to^*_{\mathcal{R}} \circledast(s_{i+1})$

  Since t_i and s_{i+1} have the same root symbol we can write $t_i = f(u_1,\dots,u_n)$ and $s_{i+1} = f(u'_1,\dots,u'_n)$. If $\circledast(f)$ is undefined or $n = 0$ then $\circledast(s_i) = s_i \to^*_{\mathcal{R}} t_i = \circledast(t_i)$. Suppose $\circledast(f) = k$. Since t_i is an instance of a right-hand side of a pair in $\mathcal{P}$ and $\circledast(\mathcal{P})$ is well-defined, u_k cannot be a variable. Write $u_k = g(v_1,\dots,v_m)$. According to $\circledast$-stability, u_k is root stable and thus $u'_k = g(v'_1,\dots,v'_m)$. Hence

  $$\begin{aligned} t_i &= f(u_1,\dots,u_{k-1},g(v_1,\dots,v_m),u_{k+1},\dots,u_n) \\ s_{i+1} &= f(u'_1,\dots,u'_{k-1},g(v'_1,\dots,v'_m),u'_{k+1},\dots,u'_n) \end{aligned}$$

  and

  $$\begin{aligned} \circledast(t_i) &= \circledast^f_g(u_1,\dots,u_{k-1},v_1,\dots,v_m,u_{k+1},\dots,u_n) \\ \circledast(s_{i+1}) &= \circledast^f_g(u'_1,\dots,u'_{k-1},v'_1,\dots,v'_m,u'_{k+1},\dots,u'_n) \end{aligned}$$

  Consequently, $t_i \to^*_{\mathcal{R}} s_{i+1}$ if and only if $u_j \to^*_{\mathcal{R}} u'_j$ for $1 \leqslant j \leqslant n$ with $j \neq k$ and $v_j \to^*_{\mathcal{R}} v'_j$ for $1 \leqslant j \leqslant m$ if and only if $\circledast(t_i) \to^*_{\mathcal{R}} \circledast(s_{i+1})$.

- t_i terminates wrt $\mathcal{R}$ if and only if $\circledast(t_i)$ terminates wrt $\mathcal{R}$

  This follows immediately from the observation above that all reductions in t_i take place in the arguments u_j or v_j. □

Corollary 40. *The DP processor $\mathcal{U}_2$ is sound and complete.* □

The next example shows that $\circledast$-stability is essential for soundness.

Example 41. Consider the non-terminating ATRS $\mathcal{R}$ consisting of the two rules $\mathsf{f}\,\mathsf{a} \to \mathsf{g}\,\mathsf{a}$ and $\mathsf{g} \to \mathsf{f}$, which induces the infinite DP problem $(\mathcal{P},\mathcal{R})$ with $\mathcal{P}$ consisting of the rules $\mathsf{f}^\sharp\,\mathsf{a} \to \mathsf{g}^\sharp\,\mathsf{a}$ and $\mathsf{f}^\sharp\,\mathsf{a} \to \mathsf{g}^\sharp$. Since $\mathcal{P}{\downarrow_{\mathcal{U}}} = \mathcal{P}$ and $\mathcal{U}_1$ is sound, the DP problem $(\mathcal{P},\mathcal{U}^+(\mathcal{R}_\eta))$ is also infinite. The set $\circledast(\mathcal{P}{\downarrow_{\mathcal{U}}})$ consists of $\mathsf{f}^\sharp_1(\mathsf{a}) \to \mathsf{g}^\sharp_1(\mathsf{a})$ and $\mathsf{f}^\sharp_1(\mathsf{a}) \to \mathsf{g}^\sharp$. Clearly, the DP problem $(\circledast(\mathcal{P}),\mathcal{U}^+(\mathcal{R}_\eta))$ is finite. Note that $(\mathcal{P},\mathcal{U}^+(\mathcal{R}_\eta))$ is not $\circledast$-stable as $\mathsf{g} \xrightarrow{\epsilon}_{\mathcal{U}^+(\mathcal{R}_\eta)} \mathsf{f}$.

Since $\circledast$-stability is undecidable in general, for automation we need to approximate strong root stability. We present a simple criterion which is based on the term approximation TCAP from [10], where it was used to give a better approximation of dependency graphs.

Definition 42 ([10]). *Let $\mathcal{R}$ be a TRS and t a term. The term $\mathsf{TCAP}_\mathcal{R}(t)$ is inductively defined as follows. If t is a variable, $\mathsf{TCAP}_\mathcal{R}(t)$ is a fresh variable. If $t = f(t_1,\dots,t_n)$ then we let $u = f(\mathsf{TCAP}_\mathcal{R}(t_1),\dots,\mathsf{TCAP}_\mathcal{R}(t_n))$ and define $\mathsf{TCAP}_\mathcal{R}(t)$ to be u if u does not unify with the left-hand side of a rule in $\mathcal{R}$, and a fresh variable otherwise.*

Lemma 43. *A term t is strongly root stable for a TRS $\mathcal{R}$ if $\mathsf{TCAP}_\mathcal{R}(t) \notin \mathcal{V}$.*

Proof. The only possibility for $\mathsf{TCAP}_\mathcal{R}(t) \notin \mathcal{V}$ is when $t = f(t_1,\dots,t_n)$ and $u = f(\mathsf{TCAP}_\mathcal{R}(t_1),\dots,\mathsf{TCAP}_\mathcal{R}(t_n))$ does not unify with a left-hand side of a rule in $\mathcal{R}$. Assume to the contrary that t is not strongly root stable. Then there are a substitution σ and a left-hand side l of a rule in $\mathcal{R}$ such that $t\sigma \xrightarrow{>\epsilon}{}^*_\mathcal{R} l\tau$. Write $l = f(l_1,\dots,l_n)$. We have $t\sigma = f(t_1\sigma,\dots,t_n\sigma)$ with $t_i\sigma \to^*_\mathcal{R} l_i\tau$ for $1 \leqslant i \leqslant n$. Hence $\mathsf{TCAP}_\mathcal{R}(t_i)\delta_i = l_i\tau$ for some substitution δ_i ([10, proof of Theorem 13]). Since the terms $\mathsf{TCAP}_\mathcal{R}(t_1),\dots,\mathsf{TCAP}_\mathcal{R}(t_n)$ are linear and do not share variables, it follows that u unifies with l, contradicting the assumption. □

Example 44. Consider the DP problem $(\mathcal{P}{\downarrow_{\mathcal{U}}},\mathcal{U}^+(\mathcal{R}_\eta))$ of Example 35 with $\mathcal{P}{\downarrow_{\mathcal{U}}} = \{\mathsf{a}_1(x) \circ^\sharp \mathsf{a} \to \mathsf{a}_1(\mathsf{a}_1(\mathsf{a})) \circ^\sharp x\}$ and $\mathcal{U}^+(\mathcal{R}_\eta) = \{\mathsf{a} \circ x \to \mathsf{a}_1(x), \mathsf{a}_1(x) \circ y \to \mathsf{a}_2(x,y), \mathsf{a}_2(x,\mathsf{a}) \to \mathsf{a}_2(\mathsf{a}_1(\mathsf{a}),x)\}$. Since $\mathsf{TCAP}_{\mathcal{U}^+(\mathcal{R}_\eta)}(\mathsf{a}_1(\mathsf{a}_1(\mathsf{a}))) = \mathsf{a}_1(\mathsf{a}_1(\mathsf{a}))$ is not a variable, $\mathsf{a}_1(\mathsf{a}_1(\mathsf{a}))$ is strongly root stable. Hence $(\mathcal{P}{\downarrow_{\mathcal{U}}},\mathcal{U}^+(\mathcal{R}_\eta))$ is $\circledast$-stable.

5 Experiments

The results of this paper are implemented in the termination prover T$_\text{T}$T$_2$.[2] For experimentation the 195 ATRSs from the termination problem data base (TPDB)[3] have been employed. All tests have been performed on a single core of a server equipped with eight dual-core AMD Opteron® processors 885 running at a clock rate of 2.6GHz and 64GB of main memory. Comprehensive details of the experiments[4] give evidence that the proposed transformations can be implemented very efficiently, e.g., for the most advanced strategy all 195 systems

[2] http://colo6-c703.uibk.ac.at/ttt2/
[3] http://www.lri.fr/~marche/tpdb/
[4] http://colo6-c703.uibk.ac.at/ttt2/uncurry/

Table 1. Experimental results

	direct			as processor			
	6	16	16+30	none	$\mathcal{A}$	$\mathcal{U}_1$	$\mathcal{U}_2$
subterm criterion	1	47	48	41	–	41	58
matrix (dimension 1)	4	90	101	66	71	95	101
matrix (dimension 2)	7	108	131	108	115	136	138

are analyzed within about 15 seconds. We considered two popular termination methods, namely the subterm criterion [13] and matrix interpretations [8] of dimensions one and two and with coefficients ranging over $\{0, 1\}$. Both methods are integrated within the dependency pair framework using dependency graph reasoning and usable rules as proposed in [10,11,12].

Table 1 differentiates between applying the transformations as a preprocessing step (direct) or within the dependency pair framework (as processor). The direct method of Corollary 6 (Theorem 16, Theorems 16 and 30) applies to 10 (141, 170) systems. If used directly, the numbers in the table refer to the systems that could be proved terminating in case of a successful transformation. Mirroring (when termination of the original system could not be proved) does increase applicability of our (direct) transformation significantly. The right part of Table 1 states the number of successful termination proofs for the processors $\mathcal{A}$ (transformation $\mathcal{A}$ from [10,17]), $\mathcal{U}_1$ (Definition 32), and $\mathcal{U}_2$ (Definition 38) which shows that the results of this paper really increase termination proving power for ATRSs. Since transformation $\mathcal{A}$ does not preserve minimality (Example 34) one cannot use it together with the subterm criterion. (In [17] it is shown that minimality is preserved when the transformation $\mathcal{A}$ is fused with the reduction pair and usable rules processors.) It is a trivial exercise to extend mirroring to DP problems. Our experiments revealed that (a) mirroring works better for the direct approach (hence we did not incorporate it into the right block of the table) and (b) the uncurrying processors should be applied before other termination processors.

Although Theorem 16 and the processor $\mathcal{U}_2$ are incomparable in power we recommend the usage of the processor. One reason is the increased strength and another one the modularity which allows to prevent pitfalls like Example 22. Last but not least, the processors $\mathcal{U}_1$ and $\mathcal{U}_2$ are not only sound but also complete which makes them suitable for non-termination analysis. Unfortunately T$_{\mathsf{T}}$T$_2$ does only support trivial methods for detecting non-termination of TRSs but we anticipate that these processors ease the job of proving non-termination of ATRSs considerably.

6 Related Work

The $\mathcal{A}$ transformation of Giesl *et al.* [10] works only on *proper* applicative DP problems, which are DP problems with the property that all occurrences of each

constant have the same number of arguments. No uncurrying rules are added to the processed DP problems. This destroys minimality (Example 34), which seriously hampers the applicability of the $\mathcal{A}$ transformation. Thiemann [17, Sections 6.2 and 6.3] addresses the loss of minimality by incorporating reduction pairs, usable rules, and argument filterings into the $\mathcal{A}$ transformation. (These refinements were considered in the column labeled $\mathcal{A}$ in Table 1.) In [17] it is further observed that the $\mathcal{A}$ transformation works better for innermost termination than for termination. A natural question for future work is how $\mathcal{U}_1$ and $\mathcal{U}_2$ behave for innermost termination.

Aoto and Yamada [1,2] present transformation techniques for proving termination of simply typed ATRSs. After performing η-saturation, head variables are eliminated by instantiating them with 'template' terms of the appropriate type. In a final step, the resulting ATRS is translated into functional form.

Example 45. Consider again the ATRS $\mathcal{R}$ of Example 7. Suppose we adopt the following type declarations: $\mathsf{0} : \mathsf{int}$, $\mathsf{s} : \mathsf{int} \to \mathsf{int}$, $\mathsf{nil} : \mathsf{list}$, $(:) : \mathsf{int} \to \mathsf{list} \to \mathsf{list}$, $\mathsf{id} : \mathsf{int} \to \mathsf{int}$, $\mathsf{add} : \mathsf{int} \to \mathsf{int} \to \mathsf{int}$, and $\mathsf{map} : (\mathsf{int} \to \mathsf{int}) \to \mathsf{list} \to \mathsf{list}$. The head variable f in the right-hand side $: (\mathsf{id}\ x)\ (\mathsf{map}\ f\ y)$ has type $\mathsf{int} \to \mathsf{int}$. There are three template terms of this type: s, id, and $\mathsf{add}\ z$. Instantiating f by these three terms in $\mathcal{R}_\eta$ produces the ATRS $\mathcal{R}'$:

$$\begin{array}{cc}
\mathsf{id}\ x \to x & \mathsf{map}\ f\ \mathsf{nil} \to \mathsf{nil} \\
\mathsf{add}\ \mathsf{0} \to \mathsf{id} & \mathsf{map}\ \mathsf{s}\ (:\ x\ y) \to :\ (\mathsf{s}\ x)\ (\mathsf{map}\ \mathsf{s}\ y) \\
\mathsf{add}\ \mathsf{0}\ y \to \mathsf{id}\ y & \mathsf{map}\ \mathsf{id}\ (:\ x\ y) \to :\ (\mathsf{id}\ x)\ (\mathsf{map}\ \mathsf{id}\ y) \\
\mathsf{add}\ (\mathsf{s}\ x)\ y \to \mathsf{s}\ (\mathsf{add}\ x\ y) & \mathsf{map}\ (\mathsf{add}\ z)\ (:\ x\ y) \to :\ (\mathsf{add}\ z\ x)\ (\mathsf{map}\ (\mathsf{add}\ z)\ y)
\end{array}$$

The TRS $\mathcal{R}'{\downarrow_{\mathcal{U}}}$ is terminating because its rules are oriented from left to right by the lexicographic path order. According to the main result of [2], the *simply typed* ATRS $\mathcal{R}$ is terminating, too.

The advantage of the simply typed approach is that the uncurrying rules are not necessary because the application symbol has been eliminated from $\mathcal{R}'{\downarrow_{\mathcal{U}}}$. This typically results in simpler termination proofs. It is worthwhile to investigate whether a version of head variable instantiation can be developed for the untyped case. We would like to stress that with the simply typed approach one obtains termination only for those terms which are simply typed. Our approach, when it works, provides termination for all terms, irrespective of *any* typing discipline. In [3] the dependency pair method is adapted to deal with simply typed ATRSs. Again, head variable instantiation plays a key role.

Applicative term rewriting is not the only model for capturing higher-order aspects. The S-expression rewrite systems of Toyama [18] have a richer structure than applicative systems, which makes proving termination often easier. Recent methods (e.g. [6,14]) use types to exploit strong computability, leading to powerful termination methods which are directly applicable to higher-order systems. In [16] strong computability is used to analyse the termination of simply typed ATRSs with the dependency pair method.

References

1. Aoto, T., Yamada, T.: Termination of simply typed term rewriting by translation and labelling. In: Nieuwenhuis, R. (ed.) RTA 2003. LNCS, vol. 2706, pp. 380–394. Springer, Heidelberg (2003)
2. Aoto, T., Yamada, T.: Termination of simply-typed applicative term rewriting systems. In: HOR 2004. Technical Report AIB-2004-03, RWTH Aachen. pp. 61–65 (2004)
3. Aoto, T., Yamada, T.: Dependency pairs for simply typed term rewriting. In: Giesl, J. (ed.) RTA 2005. LNCS, vol. 3467, pp. 120–134. Springer, Heidelberg (2005)
4. Arts, T., Giesl, J.: Termination of term rewriting using dependency pairs. Theoretical Computer Science 236(1-2), 133–178 (2000)
5. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1998)
6. Blanqui, F., Jouannaud, J.-P., Rubio, A.: HORPO with computability closure: A reconstruction. In: Dershowitz, N., Voronkov, A. (eds.) LPAR 2007. LNCS (LNAI), vol. 4790, pp. 138–150. Springer, Heidelberg (2007)
7. Dershowitz, N.: 33 Examples of termination. In: French Spring School of Theoretical Computer Science. LNCS, vol. 909, pp. 16–26. Springer, Heidelberg (1995)
8. Endrullis, J., Waldmann, J., Zantema, H.: Matrix interpretations for proving termination of rewrite systems. Journal of Automated Reasoning 40(2-3), 195–220 (2008)
9. Giesl, J., Thiemann, R., Schneider-Kamp, P.: The dependency pair framework: Combining techniques for automated termination proofs. In: Baader, F., Voronkov, A. (eds.) LPAR 2004. LNCS, vol. 3452, pp. 301–331. Springer, Heidelberg (2005)
10. Giesl, J., Thiemann, R., Schneider-Kamp, P.: Proving and disproving termination of higher-order functions. In: Gramlich, B. (ed.) FroCos 2005. LNCS (LNAI), vol. 3717, pp. 216–231. Springer, Heidelberg (2005)
11. Giesl, J., Thiemann, R., Schneider-Kamp, P., Falke, S.: Mechanizing and improving dependency pairs. Journal of Automated Reasoning 37(3), 155–203 (2006)
12. Hirokawa, N., Middeldorp, A.: Automating the dependency pair method. Information and Computation 199(1-2), 172–199 (2005)
13. Hirokawa, N., Middeldorp, A.: Tyrolean termination tool: Techniques and features. Information and Computation 205(4), 474–511 (2007)
14. Jouannaud, J.P., Rubio, A.: Polymorphic higher-order recursive path orderings. Journal of the ACM 54(1) (2007)
15. Kennaway, R., Klop, J.W., Sleep, M.R., de Vries, F.J.: Comparing curried and uncurried rewriting. Journal of Symbolic Computation 21(1), 15–39 (1996)
16. Kusakari, K., Sakai, M.: Enhancing dependency pair method using strong computability in simply-typed term rewriting. Applicable Algebra in Engineering, Communication and Computing 18(5), 407–431 (2007)
17. Thiemann, R.: The DP Framework for Proving Termination of Term Rewriting. PhD thesis, RWTH Aachen, Available as technical report AIB-2007-17 (2007)
18. Toyama, Y.: Termination of S-expression rewriting systems: Lexicographic path ordering for higher-order terms. In: van Oostrom, V. (ed.) RTA 2004. LNCS, vol. 3091, pp. 40–54. Springer, Heidelberg (2004)
19. Xi, H.: Towards automated termination proofs through freezing. In: Nipkow, T. (ed.) RTA 1998. LNCS, vol. 1379, pp. 271–285. Springer, Heidelberg (1998)
20. Zantema, H.: Termination. In: Terese (ed.) Term Rewriting Systems 2003. Cambridge Tracts in Theoretical Computer Science, vol. 55, pp. 181–259. Cambridge University Press, Cambridge (2003)

Approximating Term Rewriting Systems: A Horn Clause Specification and Its Implementation*

John P. Gallagher and Mads Rosendahl

Computer Science, Building 42.2, Roskilde University, DK-4000 Denmark
{jpg,madsr}@ruc.dk

Abstract. We present a technique for approximating the set of reachable terms of a given term rewriting system starting from a given initial regular set of terms. The technique is based on previous work by other authors with the same goal, and yields a finite tree automaton recognising an over-approximation of the set of reachable terms. Our contributions are, firstly, to use Horn clauses to specify the transitions of a possibly infinite-state tree automaton defining (at least) the reachable terms. Apart from being a clear specification, the Horn clause model is the basis for further automatic approximations using standard logic program analysis techniques, yielding finite-state tree automata. The approximations are applied in two stages: first a regular approximation of the model of the given Horn clauses is constructed, and secondly a more precise relational abstraction is built using the first approximation. The analysis uses efficient representations based on BDDs, leading to more scalable implementations. We report on preliminary experimental results.

1 Introduction

We consider the problem of automatically approximating the set of reachable terms of a term rewriting system (TRS) given a set of initial terms. The problem has been studied in several contexts such as flow analysis of higher-order functional languages [20], cryptographic protocol analysis [16] and more generally the static analysis of any programming language whose operational semantics is expressed as a TRS [2]. The applications have in common the goal of proving properties of the set of all reachable terms of the TRS, which is infinite in general. It is obviously sufficient to show that the required property holds in an over-approximation of the reachable set. Safety properties, namely assertions that some given terms are not reachable, form an important class of properties that can proved using over-approximations.

For practical application of this principle it is necessary to describe the over-approximations in some decidable formalism so that the properties can be effectively checked. Regular tree languages, described by tree grammars or finite tree

* Work supported by the Danish Natural Science Research Council project *SAFT: Static Analysis Using Finite Tree Automata*.

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 682–696, 2008.

automata, provide a suitable formalism. Thus we focus on the specific problem of deriving a regular tree language containing the set of reachable terms of a TRS, starting from a possibly infinite set of initial terms also expressed as a regular tree language. Our approach builds on work by Feuillade *et al.* [9] and also by Jones [20] and Jones and Andersen [21] (an updated version of [20]).

In Section 2 we review the basic notions concerning term rewriting systems and regular tree languages expressed as finite tree automata. Following this, Section 3 contains a Horn clause specification of a possibly infinite tree automaton over-approximating the reachable terms of a given TRS and initial set. The problem is thus shifted to computing the model of this set of Horn clauses, or some approximation of the model. In Section 4, methods of approximating Horn clause models are outlined, drawing on research in the abstract interpretation of logic programs. Section 5 explains how BDD-based methods can be used to compute the (approximate) models, enhancing the scalability of the approach. Some initial experiments are reported. Finally, Section 6 contains a discussion of related work and concludes.

2 Term Rewriting Systems and Their Approximation

A *term rewriting system* (TRS for short) is formed from a non-empty finite signature Σ and a denumerable set of variables $\mathcal{V}$. $\mathsf{Term}(\Sigma \cup \mathcal{V})$ denotes the smallest set containing $\mathcal{V}$ such that $f(t_1, \ldots, t_n) \in \mathsf{Term}(\Sigma \cup \mathcal{V})$ wherever $t_1, \ldots, t_n \in \mathsf{Term}(\Sigma \cup \mathcal{V})$ and $f \in \Sigma$ has arity n. $\mathsf{Term}(\Sigma)$ denotes the subset of $\mathsf{Term}(\Sigma \cup \mathcal{V})$ containing only variable-free (ground) terms. The set of variables in a term t is denoted $\mathsf{vars}(t)$. A *substitution* is a function $\mathcal{V} \to \mathsf{Term}(\Sigma \cup \mathcal{V})$; substitutions are naturally extended to apply to $\mathsf{Term}(\Sigma \cup \mathcal{V})$. We write substitution application in postfix form – $t\theta$ stands for the application of substitution θ to t. A term t' is a *ground instance* of term t if t' is ground and $t' = t\theta$ for some substitution θ. As for notation, variable names from $\mathcal{V}$ will start with a capital letter and elements of Σ will start with lower-case letters, or numbers.

A term rewriting system is a set of rules of the form $l \to r$, where $l, r \in \mathsf{Term}(\Sigma \cup \mathcal{V})$. In a rule $l \to r$, l is called the left-hand-side (lhs) and r the right-hand-side (rhs). We consider here TRSs formed from a finite set of rules, satisfying the conditions $l \notin \mathcal{V}$ and $\mathsf{vars}(r) \subseteq \mathsf{vars}(l)$. A left- (resp. right-) linear TRS is one in which no variable occurs more than once in the lhs (resp. rhs) of a rule. In this paper we consider only left-linear TRSs.

The operational semantics of a TRS is defined as the reflexive, transitive closure of a relation $\Rightarrow$ over $\mathsf{Term}(\Sigma) \times \mathsf{Term}(\Sigma)$. The $\Rightarrow$ relation captures the concept of a "rewrite step". Intuitively, $t_1 \Rightarrow t_2$ for $t_1, t_2 \in \mathsf{Term}(\Sigma)$ holds if there is a subterm of t_1 that is a ground instance of the lhs of some rewrite rule; t_2 is the result of replacing that subterm by the corresponding instance of the rhs. More precisely, define a *(ground) context* to be a term from $\mathsf{Term}(\Sigma \cup \{\bullet\})$ containing exactly one occurrence of $\bullet$. Let c be a context; define $c[t] \in \mathsf{Term}(\Sigma)$ to be the term resulting from replacing $\bullet$ by $t \in \mathsf{Term}(\Sigma)$. Then given a TRS, we define the relation $\Rightarrow$ as the set of pairs of the form $c[l\theta] \Rightarrow c[r\theta]$ where c is a context, $l \to r$ is a rule in the TRS and there is a substitution θ such that $l\theta$

is a ground instance of l (and hence $r\theta$ is also ground). The reflexive, transitive closure of $\Rightarrow$ is denoted $\Rightarrow^*$. Given a set of ground terms S and a TRS, the set of terms reachable from S is defined as $\mathsf{reach}(S) = \{t \mid s \in S, s \Rightarrow^* t\}$.

We are not concerned here with important aspects of TRSs such as confluence, rewrite strategies, or the distinction between constructors and defined functions. Details on TRSs can be found in the literature, for example in [8].

2.1 Finite Tree Automata

Finite tree automata (FTAs) can be seen as a restricted class of rewriting system. An FTA with signature Σ is a finite set of ground rewrite rules over an extended signature $\Sigma \cup \mathcal{Q}$ where $\mathcal{Q}$ is a set of constants (unary function symbols) disjoint from Σ, called *states*. A set $\mathcal{Q}_f \subseteq \mathcal{Q}$ is called the set of *accepting* or *final* states. Each rule is of the form $f(q_1, \ldots, q_n) \to q$ where $q_1, \ldots, q_n, q \in \mathcal{Q}$ and $f \in \Sigma$ has arity n. The relation $\Rightarrow^*$ is defined as for TRSs in general but we are interested mainly in pairs $t \Rightarrow^* q$ where $t \in \mathsf{Term}(\Sigma)$ and $q \in \mathcal{Q}_f$. In such a case we say that there is a successful *run* of the FTA for term t, or that t is *accepted* by the FTA. For a given FTA A we define $\mathcal{L}(A)$, the *term language* of A, to be $\{t \in \mathsf{Term}(\Sigma) \mid t \text{ is accepted by } A\}$.

The main point of interest of FTAs is that they define so-called *regular tree languages*, which have a number of desirable computational properties. Given an FTA A it is decidable whether $\mathcal{L}(A)$ is empty, finite or infinite, and whether a given term $t \in \mathcal{L}(A)$. Furthermore, regular tree languages are closed under union, intersection and complementation; given FTAs A_1 and A_2, we can construct an FTA $A_1 \cup A_2$ such that $\mathcal{L}(A_1 \cup A_2) = \mathcal{L}(A_1) \cup \mathcal{L}(A_2)$, and similarly for the intersection and complementation operations.

A *bottom-up deterministic* FTA, or DFTA, is one in which no two rules have the same lhs. A *complete* FTA is one in which there is a rule $f(q_1, \ldots, q_n) \to q$ for every $f \in \Sigma$ of arity n, and states $q_1, \ldots, q_n \in \mathcal{Q}$. It can be shown that for every FTA there is a complete DFTA (usually with a different set of states) accepting the same language. Further information about FTAs and regular tree languages can be found in the literature, for example in [6].

2.2 Approximation of TRSs

The set of reachable states of a TRS is not in general a regular tree language. For instance consider the TRS containing a single rule $f(X, Y) \to f(g(X), h(Y))$. The set $\mathsf{reach}(\{f(a, b)\})$ is $\{f(g^k(a), h^k(b)) \mid k > 0\}$, and it can be shown that there is no FTA that accepts precisely this set of terms. However, given a TRS and an initial set of terms S there is always an FTA A such that $\mathcal{L}(A) \supseteq \mathsf{reach}(S)$. In such a situation $\mathcal{L}(A)$ is called an *over-approximation* of $\mathsf{reach}(S)$. There could of course be more than one possible FTA defining an over-approximation and generally we prefer the most *precise*, that is, smallest over-approximations. Here, as usual in static analysis problems, there is a trade-off between complexity and precision; the more precise the approximation, the more expensive it tends to be to construct it.

There are a number of cases where an FTA gives a perfectly precise approximation, that is, there exists an FTA A such that $\mathcal{L}(A) = \mathsf{reach}(S)$. These cases

Term Rewriting System	FTA defining initial terms
$plus(0, X) \to X.$	$even(qpo) \to qf.$
$plus(s(X), Y) \to s(plus(X, Y)).$	$even(qpe) \to qf.$
$even(0) \to true.$	$s(qeven) \to qodd.$
$even(s(0)) \to false.$	$s(qodd) \to qeven.$
$even(s(X)) \to odd(X).$	$plus(qodd, qodd) \to qpo.$
$odd(0) \to false.$	$plus(qeven, qeven) \to qpe.$
$odd(s(0)) \to true.$	$0 \to qeven.$
$odd(s(X)) \to even(X).$	(Accepting state is qf)

Fig. 1. A TRS and an FTA (from [9]) defining the initial terms in Example 1

are often characterised by syntactic conditions on the TRS and the initial set S. Discussion of various classes for which this holds is contained in [19,6,9].

Example 1. To illustrate these concepts we use the TRS in Figure 1 taken from [9]. The TRS defines the operation $plus$ on natural numbers in successor notation (i.e. n represented by $s^n(0)$), and the predicates $even$ and odd on natural numbers. The FTA (call it A) defines the set of "calls" $even(plus(n_1, n_2))$ where n_1, n_2 are either both even or both odd. It can be checked that, for example $even(plus(s(0), s(s(s(0))))) \Rightarrow^* qf$ in the FTA, meaning that it is contained in the set of initial terms, and that $even(plus(s(0), s(s(s(0))))) \Rightarrow^* true$ in the TRS. The property to be proved here is that $false \notin \mathsf{reach}(\mathcal{L}(A))$, in other words, that the sum of two even or two odd numbers is not odd.

This example happens to be one where the set $\mathsf{reach}(\mathcal{L}(A))$ is precisely expressible as an FTA. The procedure given in [9] can compute this FTA and check that $false$ is not accepted by it, thus proving the required property. (As we will see our method can also achieve this).

3 Horn Clause Specification of Reachable Term Approximations

In this section we present a procedure for constructing an over-approximating FTA, given a term rewriting system R and an initial set of terms S. We assume that R is left-linear and that S is regular tree language specified by an FTA.

Our procedure is based initially on the one given in [9] and is also inspired by [20]. However unlike these works we express the procedure as a set of Horn clauses in such as way as to allow the construction of a tree automaton with an infinite set of states. In other words, the Horn clauses have a possible infinite model. Following this, we draw on existing techniques for approximating Horn clause models to produce an FTA approximation. We argue that this is a flexible and scalable approach, which exploits advances made in the analysis of logic programs.

3.1 Definite Horn Clauses

We first define definite Horn clauses and their semantics. Definite Horn clauses form a fragment of first-order logic. Let Σ be a set of function symbols, $\mathcal{V}$ a

set of variables and $\mathcal{P}$ be a set of predicate symbols. An *atomic formula* is an expression of the form $p(t_1, \ldots, t_n)$ where $p \in \mathcal{P}$ is an n-ary predicate symbol and $t_1, \ldots, t_n \in \mathsf{Term}(\Sigma \cup \mathcal{V})$. A definite Horn clause is a logical formula of the form $\forall((u_1 \wedge \ldots \wedge u_m) \rightarrow u)$ $(m \geq 0)$ where $u_1, \ldots, u_m, u$ are atomic formulas. If $m = 0$ the clause is called a *fact* or *unit clause*. A fact is often written as $true \rightarrow u$. The symbol $\forall$ indicates that all variables occurring in the clause are universally quantified over the whole formula.

From now on we will write Horn clauses "backwards" as is the convention in logic programs, and use a comma instead of the conjunction $\wedge$. The universal quantifiers are also implicit. Thus a Horn clause is written as $u \leftarrow u_1, \ldots, u_m$ or $u \leftarrow true$ in the case of facts.

The semantics of a set of Horn clauses is obtained using the usual notions of interpretation and model from classical logic. An *interpretation* is defined by (i) a domain of interpretation which is a non-empty set D; (ii) a *pre-interpretation* which is a function mapping each n-ary function $f \in \Sigma$ to a function $D^n \rightarrow D$; and (iii) a predicate interpretation which assigns to each n-ary predicate in $\mathcal{P}$ a relation in D^n. A model of a set of Horn clauses is an interpretation that satisfies each clause (using the usual notion of "satisfies" based on the meanings of the logic connectives and quantifiers; see for example [22] or any textbook on logic).

The *Herbrand interpretation* is of special significance. The domain D of a Herbrand interpretation is $\mathsf{Term}(\Sigma)$; the pre-interpretation maps each n-ary function f to the function $\hat{f} : \mathsf{Term}(\Sigma)^n \rightarrow \mathsf{Term}(\Sigma)$ defined by $\hat{f}(t_1, \ldots, t_n) = f(t_1, \ldots, t_n)$. A Herbrand interpretation of the predicates assigns each n-ary predicate a relation in $\mathsf{Term}(\Sigma)^n$. Such a relation can be conveniently represented as a set of ground atomic formulas $p(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n \in \mathsf{Term}(\Sigma)$. There exists a *least* Herbrand model (which may be infinite) of a set of definite Horn clauses, which is the most concrete interpretation; it contains exactly those ground atomic formulas that are true in every interpretation.

As will be discussed in Section 4, models other than Herbrand models are of interest for the purpose of abstracting the meaning of a set of Horn clauses.

Horn Clause representation of FTAs A simple Horn Clause representation of an FTA with signature Σ and set of states $\mathcal{Q}$ can be obtained by regarding the rewrite arrow $\rightarrow$ as a binary predicate and $\Sigma \cup \mathcal{Q}$ as the signature of the Horn clause language. An FTA rule $f(q_1, \ldots, q_n) \rightarrow q$ is thus a unit Horn clause or fact written as $(f(q_1, \ldots, q_n) \rightarrow q) \leftarrow true$.

3.2 Tree Automata Approximation of Reachable Terms

We now return to the problem of computing an FTA over-approximating the set of reachable terms of a left-linear TRS. The approach taken in [9] is based on the notion of *completion*. Intuitively, this works as follows. Let $l \rightarrow r$ be a rule in the TRS and suppose that I is the FTA defining the set of initial terms. Then the FTA A defining $\mathsf{reach}(\mathcal{L}(I))$ has to be such that

(i) $\mathcal{L}(A) \supseteq \mathcal{L}(I)$ and

(ii) if there is some ground instance of l, say $l\theta$, where θ is a substitution mapping variables to FTA states, such that $l\theta \Rightarrow^* q$ in A, for some FTA (not necessarily final) state q then $r\theta \Rightarrow^* q$ should also hold in A.

We illustrate the principle referring to Example 1. Consider the second rule of the TRS, $plus(s(X), Y) \rightarrow s(plus(X, Y))$. There exists a substitution θ, namely $\{X = qeven, Y = qodd\}$ such that $plus(s(X), Y)\theta \Rightarrow^* qpo$ using the initial FTA. Hence by requirement (ii) the FTA for the reachable terms should be such that $s(plus(X, Y))\theta \Rightarrow^* qpo$. This could be ensured by adding the rules $plus(qeven, qodd) \rightarrow q_0, s(q_0) \rightarrow qpo$, for some state q_0.

We now show how we can construct a set of Horn clauses H_A capturing requirements (i) and (ii). The Horn clauses will be such that their minimal Herbrand model will be a set of atomic formulas $f(q_1, \ldots, q_n) \rightarrow q$ defining the rules of the required FTA. First of all, H_A contains the clauses corresponding to the rules in I, which ensures condition (i) above.

Secondly, consider requirement (ii). Given a TRS rule $l \rightarrow r$, we construct Horn clauses as follows. Define the operation $\mathsf{flatlhs}(t \rightarrow Y)$ as follows, where $t \in \mathsf{Term}(\Sigma \cup \mathcal{V})$ and $Y \in \mathcal{V}$:

- $\mathsf{flatlhs}(t \rightarrow Y) = \{t \rightarrow Y\}$, if t has no non-variable proper subterms;
- $\mathsf{flatlhs}(t \rightarrow Y) = \mathsf{flatlhs}(t' \rightarrow Y) \cup \{f(X_1, \ldots, X_n) \rightarrow Y'\}$, if $f(X_1, \ldots, X_n)$ $(n \geq 0)$ is a proper subterm of t whose arguments are all variables, Y' is a fresh variable and t' is the result of replacing the subterm $f(X_1, \ldots, X_n)$ by Y' in t.

Example 2. Consider the rule $even(s(0)) \rightarrow false$. Then $\mathsf{flatlhs}(even(s(0)) \rightarrow Y_0)$ is $\{0 \rightarrow Y_1, s(Y_1) \rightarrow Y_2, even(Y_2) \rightarrow Y_0\}$.

The following claim establishes that the flattened form can be used to check the condition that there exists a substitution such that $l\theta \Rightarrow^* q$.

Claim. Let t be a linear term, A an FTA and Y a variable not occurring in t. Then there is a substitution θ mapping variables of t to FTA states such that $t\theta \Rightarrow^* q$ for some FTA state q if and only if the conjunction $\wedge(\mathsf{flatlhs}(t \rightarrow Y))$ is satisfiable in the set of FTA rules.

Now we come to the critical aspect; how to ensure that $r\theta \Rightarrow^* q$ whenever $l\theta \Rightarrow^* q$. The key question is how to choose the states in the intermediate steps of the derivation that is to be constructed (e.g. the state q_0 above). This question is discussed in detail in [9]. Essentially, in order not to lose precision, a new set of states should be created unique to that derivation, that is, depending only on the rule $l \rightarrow r$ and the substitution θ. However, when applied repeatedly to the same rule this strategy can lead to the creation of an infinite number of new states. An *abstraction* function is introduced in [9], whose purpose is to specify how to introduce a finite set of states in order to satisfy requirement (ii). The question of finding the "right" abstraction function becomes crucial.

Example 3. Consider the TRS rule $f(X) \rightarrow f(g(X))$. Suppose the initial FTA contains a rule $f(q_0) \rightarrow q_1$. Applying the completion principle, we find the

substitution $\theta_0 = \{X = q_0\}$ such that $f(X)\theta_0 \Rightarrow^* q_1$. To build the required derivation $f(g(q_0)) \Rightarrow^* q_1$ we create a new state q_2 and add the rules $g(q_0) \to q_2$ and $f(q_2) \to q_1$. Applying the completion again we find a substitution $\theta_1 = \{X = q_2\}$ such that $f(X)\theta_1 \Rightarrow^* q_1$, and then try to construct a derivation $f(g(q_2)) \Rightarrow^* q_1$. If we choose the same intermediate state q_2 we add unintended derivations (such as $g(g(q_2)) \Rightarrow^* q_2$). Therefore we pick a fresh state q_3 and add the rules $g(q_2) \to q_3$ and $f(q_3) \to q_1$. Clearly this process continues indefinitely and an infinite number of states would be added. A solution using an abstraction function, as in [9], might indeed pick the same state q_2 in both completion steps, but at the cost of possibly adding non-reachable terms.

Our approach is different; we defer the decision on how to abstract the states and define a construction that can introduce an infinite number of states. Given a TRS rule $l \to r$, associate with each occurrence of a non-variable proper subterm w of r a unique function symbol, say q_w, whose arity is the number of distinct variables within w. Define $\mathsf{flatrhs}(t \to Y)$ as follows.

- $\mathsf{flatrhs}(t \to Y) = \{t \to Y\}$, if t has no non-variable proper subterms except possibly "state" terms of form $q_{w'}(\bar{Z})$;
- $\mathsf{flatrhs}(t \to Y) = \mathsf{flatlhs}(t' \to Y) \cup \{f(s_1, \ldots, s_n) \to q_w(\bar{Z})\}$, if $f(s_1, \ldots, s_n)$ $(n \geq 0)$ is a proper subterm of t whose arguments s_i are all either variables or state terms, q_w is the function symbol associated with that subterm, $\bar{Z}$ is the tuple of distinct variables in the subterm, and t' is the result of replacing the subterm $f(s_1, \ldots, s_n)$ by $q_w(\bar{Z})$ in t.

The terms $q_w(\bar{Z})$ represent newly created FTA states. The arguments $\bar{Z}$ are substituted during the completion step and thus the states are unique to their position in r and the substitution θ associated with a completion step.

Example 4. Consider the rule in Example 3 above. Associate the unary function q_2 with the subterm $g(X)$ of the rhs. Then $\mathsf{flatrhs}(f(g(X)) \to Y_0)$ is $\{g(X) \to q_2(X), f(q_2(X)) \to Y_0\}$.

Now, the completion step simply adds new rules corresponding to the flattened form of the rhs.

Example 5. Again, considering Example 3, the first step adds the rules $g(q_0) \to q_2(q_0)$ and $f(q_2(q_0)) \to q_1$. The second step adds rules $g(q_2(q_0)) \to q_2(q_2(q_0))$ and $f(q_2(q_2(q_0))) \to q_1$, and so on.

If the rhs of a rule is a variable, the procedure flatrhs does not apply. In such cases, we replace the rule with a set of rules, one for each n-ary function f in Σ, in which the variable is substituted throughout in both lhs and rhs by a term $f(Z_1, \ldots, Z_n)$, where $Z_1, \ldots, Z_n$ are fresh distinct variables.

Example 6. The rule $plus(0, X) \to X$ in Figure 1 is replaced by the rules $plus(0, 0) \to 0$, $plus(0, s(Z)) \to s(Z)$, $plus(0, plus(Z_1, Z_2)) \to plus(Z_1, Z_2)$, and so on for each function in the signature.

```
odd(B)->D           :- 0->A, odd(B)->C, plus(A,C)->D.
false->C            :- 0->A, false->B, plus(A,B)->C.
true->C             :- 0->A, true->B, plus(A,B)->C.
even(B)->D          :- 0->A, even(B)->C, plus(A,C)->D.
s(B)->D             :- 0->A, s(B)->C, plus(A,C)->D.
0->C                :- 0->A, 0->B, plus(A,B)->C.
plus(B,C)->E        :- 0->A, plus(B,C)->D, plus(A,D)->E.
plus(A,C)->q0(A,C)  :- s(A)->B, plus(B,C)->D.
s(q0(A,C))->D       :- s(A)->B, plus(B,C)->D.
true->B             :- 0->A, even(A)->B.
false->C            :- 0->A, s(A)->B, even(B)->C.
odd(A)->C           :- s(A)->B, even(B)->C.
false->B            :- 0->A, odd(A)->B.
true->C             :- 0->A, s(A)->B, odd(B)->C.
even(A)->C          :- s(A)->B, odd(B)->C.

%Initial FTA rules
even(qpo)->qf           :- true.      plus(qodd,qodd)->qpo    :- true.
even(qpe)->qf           :- true.      plus(qeven,qeven)->qpe  :- true.
s(qeven)->qodd          :- true.      0->qeven                :- true.
s(qodd)->qeven          :- true.
```

Fig. 2. Horn Clauses for the TRS and FTA in Figure 1

The Horn clauses specifying the completion can now be constructed. For each rule $l \to r$ in the TRS (with variable rhs rules replaced as just shown) let Q be a variable not occurring in the rule. Construct the set of Horn clauses $\{H \leftarrow \wedge(\mathsf{flatlhs}(l \to Q)) \mid H \in \mathsf{flatrhs}(r \to Q)\}$.

Example 7. Consider the rule $plus(s(X), Y) \to s(plus(X, Y))$ from Example 1. Let q_0 be the unique function symbol associated with the subterm $plus(X, Y)$ of the rhs. Then the two Horn clauses for the rule are:

$$(plus(X, Y) \to q_0(X, Y)) \leftarrow (s(X) \to Q_1), (plus(Q_1, Y) \to Q).$$
$$(s(q_0(X, Y)) \to Q) \leftarrow (s(X) \to Q_1), (plus(Q_1, Y) \to Q).$$

These rules, together with all the others for the TRS and FTA in Figure 1, are shown in Figure 2. Note that the first seven clauses correspond to the instances of the rule $plus(0, X) \to X$.

The minimal Herbrand model of a Horn clause program constructed in this way contains a set of tree automaton rules. If the model is finite it represents an FTA A, whose states are teh states occurring in the rules and whose final states are the same as those of the initial FTA I (which is a subset of A). In this case we claim that $\mathcal{L}(A) \supseteq \mathsf{reach}(\mathcal{L}(I))$. The Horn clause program directly captures the requirements (i) and (ii) discussed at the start of Section 3.2. Furthermore, conditions under which $\mathcal{L}(A) = \mathsf{reach}(\mathcal{L}(I))$ can be obtained using the formal techniques in [9] as a guide, but this is beyond the scope of this paper. Our primary interest is in handling the case where the model is infinite and therefore not an FTA.

Even where A is infinite, the $\Rightarrow^*$ and the set $\mathcal{L}(A)$ are defined; $\mathcal{L}(A)$ contains $\mathsf{reach}(\mathcal{L}(I))$, though $\mathcal{L}(A)$ is no longer a regular language. The problem of reasoning about the reachable terms $\mathsf{reach}(\mathcal{L}(I))$ is now shifted to the problem of reasoning about $\mathcal{L}(A)$. In our approach, we use static analysis techniques developed for approximating Horn clauses models.

4 Tree Automata Approximations of Horn Clause Models

The minimal Herbrand model of a definite Horn clause program P can be computed as the least fixed point of an immediate consequences operator T_P; the least fixed point is the limit of the sequence $\{T_P^n(\emptyset)\}$, $n = 0, 1, 2, \ldots$ [22]. In general this sequence does not terminate because the model of P is infinite.

As it happens the program in Figure 2 has a finite model and hence the sequence converges to a limit in a finite number of steps. Furthermore the minimal Herbrand model thus obtained contains no rule with lhs *false*, thus proving the required property since *false* cannot possibly be reachable.

However, in general the fixpoint construction does not terminate, as already noted. In this case we turn to general techniques for approximating the models of Horn clauses, developed in the field of static analysis of logic programs. Such methods are mostly based on abstract interpretation [7].

4.1 FTA Approximation of the Minimal Model

The problem of computing regular tree language approximations of logic programs has its roots in automatic type inference [23,5,18]. Later, various authors developed techniques for computing tree grammar approximations of the model of a logic program [17,10,12,26,27,15]. Approaches based on solving set constraints [17] can be contrasted with those computing an abstract interpretation over a domain of tree grammars, but all of these have in common that they derive a tree grammar in some form, whose language over-approximates the model of the analysed program. The most precise of these techniques, such as [17,10,15] compute a general FTA (rather than a top-down deterministic tree grammar).

These techniques (specifically, the implementation based on [15]) have been applied to our Horn clause programs defining the reachable terms in a TRS. This yields an FTA describing a superset of the model of the Horn clause program. More precisely, let P be a Horn clause program and $M[\![P]\!]$ be the minimal Herbrand model of P. We derive an FTA A_P such that $\mathcal{L}(A_P) \supseteq M[\![P]\!]$.

In our case, $M[\![P]\!]$ contains the rules of a tree automaton and the approximating FTA A_P describes a superset of $M[\![P]\!]$; that is, $\mathcal{L}(A_P)$ is a set of *rules*. (It is important not to confuse the FTA derived by regular tree approximation of the Horn clause model from the tree automaton represented by the model itself). A_P could in principle be used itself in order to reason about reachability properties, but the possibilities here seem to be limited to checking whether rules of a certain syntactic form are present in the model. The main purpose of the approximation A_P is as a stepping stone for deriving a more precise approximation and obtaining an FTA approximation of $\mathsf{reach}(\mathcal{L}(I))$.

Term Rewriting System

$plus(0, X) \rightarrow X.$	$even(s(X)) \rightarrow odd(X).$
$plus(s(X), Y) \rightarrow s(plus(X, Y)).$	$odd(0) \rightarrow false.$
$times(0, X) \rightarrow 0.$	$odd(s(0)) \rightarrow true.$
$times(s(X), Y) \rightarrow plus(Y, times(X, Y)).$	$odd(s(X)) \rightarrow even(X).$
$square(X) \rightarrow times(X, X).$	$even(square(X)) \rightarrow odd(square(s(X))).$
$even(0) \rightarrow true.$	$odd(square(X)) \rightarrow even(square(s(X))).$
$even(s(0)) \rightarrow false.$	

FTA defining initial terms

$even(s1) \rightarrow s0.$	$square(s2) \rightarrow s1.$
$0 \rightarrow s2.$	

Fig. 3. Another TRS and an FTA (from [9])

4.2 Relational Abstract Interpretation Based on an FTA

In [13] it was shown that an arbitrary FTA could be used to construct an abstract interpretation of a logic program. The process has the following steps: (i) construct an equivalent *complete deterministic FTA* (or DFTA) from the given FTA; define a finite *pre-interpretation* whose domain is the set of states of the DFTA; (iii) compute the least model with respect to that pre-interpretation using a terminating iterative fixpoint computation. The general approach of using pre-interpretations as abstractions was developed in [3,4] and a practical framework and experiments establishing its practicality were described in [11].

The abstract models derived from FTAs in this way are *relational abstractions*; they are optimal in the sense that they are the most precise models based on the derived pre-interpretation. The computed model is in general more precise than the language of the original FTA from which the pre-interpretation is constructed. A simple example illustrates the kind of precision gain. Given the standard program for appending lists

```
append([],Ys,Ys).    append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
```

the FTA approximation methods cited yield the rules $\{append(list, any, any) \rightarrow append\_type, [] \rightarrow list, [any|list] \rightarrow list\}$ together with some rules defining *any*. It can be seen that no information is returned about the second and third arguments of *append*. The pre-interpretation obtained by determinising this FTA has a domain $\{list, nonlist\}$ with interpretation $\mathcal{D}$ given by

$$\mathcal{D}([]) = list, \mathcal{D}([list|list]) = \mathcal{D}([nonlist|list]) = list,$$
$$\mathcal{D}([list|nonlist] = \mathcal{D}([nonlist|nonlist] = nonlist.$$

The minimal model with respect to this pre-interpretation is the relation

$$\{append(list, list, list), append(list, nonlist, nonlist)\}$$

which is more precise, containing relational information on the dependency between the second and third arguments. Note that no further information is

```
odd(B)->D             :- 0->A, odd(B)->C, plus(A,C)->D.
false->C              :- 0->A, false->B, plus(A,B)->C.
true->C               :- 0->A, true->B, plus(A,B)->C.
even(B)->D            :- 0->A, even(B)->C, plus(A,C)->D.
square(B)->D          :- 0->A, square(B)->C, plus(A,C)->D.
times(B,C)->E         :- 0->A, times(B,C)->D, plus(A,D)->E.
s(B)->D               :- 0->A, s(B)->C, plus(A,C)->D.
0->C                  :- 0->A, 0->B, plus(A,B)->C.
plus(B,C)->E          :- 0->A, plus(B,C)->D, plus(A,D)->E.
plus(A,C)->q0(A,C)    :- s(A)->B, plus(B,C)->D.
s(q0(A,C))->D         :- s(A)->B, plus(B,C)->D.
0->C                  :- 0->A, times(A,B)->C.
times(A,C)->q1(A,C)   :- s(A)->B, times(B,C)->D.
plus(C,q1(A,C))->D    :- s(A)->B, times(B,C)->D.
times(A,A)->B         :- square(A)->B.
true->B               :- 0->A, even(A)->B.
false->C              :- 0->A, s(A)->B, even(B)->C.
odd(A)->C             :- s(A)->B, even(B)->C.
false->B              :- 0->A, odd(A)->B.
true->C               :- 0->A, s(A)->B, odd(B)->C.
even(A)->C            :- s(A)->B, odd(B)->C.
s(A)->q2(A)           :- square(A)->B, even(B)->C.
square(q2(A))->q3(A)  :- square(A)->B, even(B)->C.
odd(q3(A))->C         :- square(A)->B, even(B)->C.
s(A)->q4(A)           :- square(A)->B, odd(B)->C.
square(q4(A))->q5(A)  :- square(A)->B, odd(B)->C.
even(q5(A))->C        :- square(A)->B, odd(B)->C.
even(s1)->s0          :- true.
square(s2)->s1        :- true.
0->s2                 :- true.
```

Fig. 4. Horn Clauses for the TRS and FTA in Figure 3

supplied beyond the original FTA; the analysis just exploits more fully the information available in the FTA.

Let P be a Horn clause program constructed from a TRS, as shown in Section 3. A pre-interpretation $\mathcal{D}$ maps each state of the possibly infinite automaton in the model $M[\![P]\!]$ onto one of the finite number of domain elements of $\mathcal{D}$. Let $f(q_1,\ldots,q_n) \to q \in M[\![P]\!]$. $\mathcal{D}$ is extended to rules as follows: $\mathcal{D}(f(q_1,\ldots,q_n) \to q) = f(\mathcal{D}(q_1),\ldots,\mathcal{D}(q_n)) \to \mathcal{D}(q)$. Define the FTA $M^{\mathcal{D}}[\![P]\!] = \{\mathcal{D}(l \to r) \mid l \to r \in M[\![P]\!]\}$. It is trivial to show that $\mathcal{L}(M^{\mathcal{D}}[\![P]\!]) \supseteq \mathcal{L}(M[\![P]\!])$, and hence $\mathcal{L}(M^{\mathcal{D}}[\![P]\!]) \supseteq \mathsf{reach}(\mathcal{L}(I))$.

Example 8. Consider the example in Figure 3, also taken from [9]. This example is intended to show that the square of any even (resp. odd) number is even (resp. odd). In order to prove this it is sufficient to show that *false* is not reachable from the given initial FTA. Applying the Horn clause construction we obtain the program in Figure 4. The model of this program is not finite. We compute

an FTA approximation of this program using the method of [15]. The output FTA is automatically refined by splitting all non-recursive states as follows. Let q be a non-recursive state having rules $l_1 \to q, \ldots, l_k \to q$ $(k > 1)$ in the FTA. Introduce fresh states $q^1, \ldots, q^k$ and replace the rules for q with rules $l_1 \to q^1, \ldots, l_k \to q^k$. Then replace each rule containing q by k copies with q replaced by $q^1, \ldots, q^k$ respectively. This transformation (which is optional) preserves the language of the automaton and gives a more fine-grained abstraction. Following this we construct a pre-interpretation containing 53 elements which are the states of the determinised transformed FTA and compute the abstract model of the Horn clause program. The model contains no rule $false \to s0$, thus proving the required property. Compare this automatic approach (no information is supplied beyond the TRS and the initial FTA) with the semi-automatic method described in [9] in which an abstraction is introduced on the fly with no obvious motivation for the particular abstraction chosen.

Other approximation techniques are available apart from those illustrated in Example 8. For example, a pre-interpretation $\mathcal{D}_{pos}$ can be defined as $\mathcal{D}_{pos}(q(\_, \ldots, \_) = q$; this simply abstracts each state by the identifier of the position in the TRS where it originated, ignoring the substitution θ. The result is closely related to the approximation obtained by Jones [20].

5 BDD-Based Implementation

In [14] it was shown that static analysis of logic programs using abstraction based on FTAs could be implemented using BDD-based techniques. The main components of this method are (i) a generic tool for computing the model of a Datalog program [25] and (ii) a compact representation (*product form*) of the determinised FTA used to construct the pre-interpretation. Determinisation can cause exponential blow-up in the number of states and rules, but our experience is that the number of states remains manageable. The number of rules can blow up in any case but using the product form often yields orders of magnitude reduction in the representation size. The product form can also be represented as a Datalog program.

(Definite) Datalog programs consist of Horn clauses containing no function symbols of arity more than zero. Our Horn clauses contain such functions but they are easily transformed away; an atomic formula of the form $f(X_1, \ldots, X_n) \to Y$ is transformed to $rule\_f(X_1, \ldots, X_n, Y)$ which conforms to the Datalog syntax. Details of how to represent the pre-interpretation and the determinised FTA as Datalog clauses can be found in [14].

Preliminary experiments using examples from [9,1] indicate that the BDD-based tool gives substantial speedup compared to other approaches. The `evenodd` and `evensq` examples are those in Figures 1 and 3. `smart`, `combi` and `nspk` were kindly supplied by the authors of [1], where they are described in more detail. `smart` and `nspk` are cryptographic protocol models, while `combi` is simply a combinatorial example which produces a large FTA. In Figure 5, P gives the number of clauses in the Horn clause program, the two $\mathcal{D}$ columns give the size

Name	P	$\mathcal{D}$ size (states)	$\mathcal{D}$ time (msecs)	model (msecs)	proof
`evenodd`	22	17	20	50	✓
`evensq`	30	53	66	179	✓
`smart`	123	97 (pos)	8	3270	✓
`combi`	44	25	54	38	n/a
`nspk`	56	183	1782	1170	×

Fig. 5. Experimental results

of the pre-interpretation and the time required to generate it, and the next column shows the time taken to compute the model. The last column indicates whether the required property was proved. In the case of `nspk` the property was not proved in the abstract model. The generation of the pre-interpretation for `smart` as described in Section 4 exhausted memory; however the simpler pre-interpretation ($\mathcal{D}_{pos}$, see Section 4) was sufficient to prove the required property. Timings were taken on a machine with a dual-core Athlon 64-bit processor, with 4GByte RAM, and the tools are implemented in Ciao Prolog and the `bddbddb` tool developed by Whaley [29,28]. These results indicate that the model computation appears to scale well, but further research on generating useful pre-interpretation is needed, and comparing with the strategies for generating abstractions in other approaches, especially the Timbuk system [16].

6 Discussion and Conclusions

The most closely related work to ours, as already mentioned, is that of Feuillade *et al.* [9]. That research has yielded impressive experimental results and extensive analysis of the various classes of TRS that have exact solutions. A tool (Timbuk [16]) has been developed to support the work. Our approach contrasts with that work by being completely automatic, relying more on established analysis techniques based on abstract interpretation, and by permitting a BDD-based implementation that should give greater scalability.

The work of Jones and Andersen [20,21] pre-dates the above work though the authors of [9] seem to be unaware of it. It has some similarities with the above work, in that it is based on a completion procedure. It differs in allowing a more flexible tree grammar to represent the reachable terms. However this flexibility leads to a more complex completion procedure. The set of FTA states (tree grammar non-terminals) is fixed in their procedure. Their work is concerned with flow analysis and follows in the footsteps of tree grammar approximation going back to Reynolds [24].

The next stage in our research is to experiment with the various available regular tree approximation algorithms for Horn clauses, and the generation of pre-interpretations from the resulting FTAs. Following this we hope to perform substantial experiments on the scalability and precision of our techniques, building on the promising application on smaller examples so far.

Acknowledgements. We thank the anonymous referees for LPAR 2008 for detailed and helpful comments.

References

1. Balland, E., Boichut, Y., Moreau, P.-E., Genet, T.: Towards an efficient implementation of tree automata completion. In: Meseguer, J., Roşu, G. (eds.) AMAST 2008. LNCS, vol. 5140, pp. 67–82. Springer, Heidelberg (2008)
2. Boichut, Y., Genet, T., Jensen, T.P., Roux, L.L.: Rewriting approximations for fast prototyping of static analyzers. In: Baader, F. (ed.) RTA 2007. LNCS, vol. 4533, pp. 48–62. Springer, Heidelberg (2007)
3. Boulanger, D., Bruynooghe, M.: A systematic construction of abstract domains. In: LeCharlier, B. (ed.) SAS 1994. LNCS, vol. 864, pp. 61–77. Springer, Heidelberg (1994)
4. Boulanger, D., Bruynooghe, M., Denecker, M.: Abstracting *s*-semantics using a model-theoretic approach. In: Hermenegildo, M., Penjam, J. (eds.) PLILP 1994. LNCS, vol. 844, pp. 432–446. Springer, Heidelberg (1994)
5. Bruynooghe, M., Janssens, G.: An instance of abstract interpretation integrating type and mode inferencing. In: Kowalski, R., Bowen, K. (eds.) Proceedings of ICLP/SLP, pp. 669–683. MIT Press, Cambridge (1988)
6. Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree Automata Techniques and Applications (1999), http://www.grappa.univ-lille3.fr/tata
7. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM Symposium on Principles of Programming Languages, Los Angeles, pp. 238–252 (1977)
8. Dershowitz, N., Jouannaud, J.-P.: Rewrite systems. In: Handbook of Theoretical Computer Science, Formal Models and Semantics, vol B, pp. 243–320. Elsevier and MIT Press (1990)
9. Feuillade, G., Genet, T., Tong, V.V.T.: Reachability analysis over term rewriting systems. J. Autom. Reasoning 33(3-4), 341–383 (2004)
10. Frühwirth, T., Shapiro, E., Vardi, M., Yardeni, E.: Logic programs as types for logic programs. In: Proceedings of the IEEE Symposium on Logic in Computer Science, Amsterdam (July 1991)
11. Gallagher, J.P., Boulanger, D., Sağlam, H.: Practical model-based static analysis for definite logic programs. In: Lloyd, J.W. (ed.) Proc. of International Logic Programming Symposium, pp. 351–365. MIT Press, Cambridge (1995)
12. Gallagher, J.P., de Waal, D.: Fast and precise regular approximation of logic programs. In: Van Hentenryck, P. (ed.) Proceedings of the International Conference on Logic Programming (ICLP 1994), Santa Margherita Ligure. MIT Press (1994)
13. Gallagher, J.P., Henriksen, K.S.: Abstract domains based on regular types. In: Demoen, B., Lifschitz, V. (eds.) ICLP 2004. LNCS, vol. 3132, pp. 27–42. Springer, Heidelberg (2004)
14. Gallagher, J.P., Henriksen, K.S., Banda, G.: Techniques for scaling up analyses based on pre-interpretations. In: Gabbrielli, M., Gupta, G. (eds.) ICLP 2005. LNCS, vol. 3668, pp. 280–296. Springer, Heidelberg (2005)
15. Gallagher, J.P., Puebla, G.: Abstract interpretation over non-deterministic finite tree automata for set-based analysis of logic programs. In: Krishnamurthi, S., Ramakrishnan, C.R. (eds.) PADL 2002. LNCS, vol. 2257. Springer, Heidelberg (2002)

16. Genet, T., Tong, V.V.T.: Reachability analysis of term rewriting systems with *timbuk*. In: Nieuwenhuis, R., Voronkov, A. (eds.) LPAR 2001. LNCS, vol. 2250, pp. 695–706. Springer, Heidelberg (2001)
17. Heintze, N., Jaffar, J.: A Finite Presentation Theorem for Approximating Logic Programs. In: Proceedings of the 17th Annual ACM Symposium on Principles of Programming Languages, pp. 197–209. ACM Press, San Francisco (1990)
18. Horiuchi, K., Kanamori, T.: Polymorphic type inference in Prolog by abstract interpretation. In: Proc. 6th Conference on Logic Programming. LNCS, vol. 315, pp. 195–214. Springer, Heidelberg (1987)
19. Jacquemard, F.: Decidable approximations of term rewriting systems. In: Ganzinger, H. (ed.) RTA 1996. LNCS, vol. 1103, pp. 362–376. Springer, Heidelberg (1996)
20. Jones, N.: Flow analysis of lazy higher order functional programs. In: Abramsky, S., Hankin, C. (eds.) Abstract Interpretation of Declarative Languages, Ellis-Horwood (1987)
21. Jones, N.D., Andersen, N.: Flow analysis of lazy higher-order functional programs. Theor. Comput. Sci. 375(1-3), 120–136 (2007)
22. Lloyd, J.: Foundations of Logic Programming, 2nd edn. Springer, Heidelberg (1987)
23. Mishra, P.: Towards a theory of types in Prolog. In: Proceedings of the IEEE International Symposium on Logic Programming (1984)
24. Reynolds, J.C.: Automatic construction of data set definitions. In: Morrell, J. (ed.) Information Processing, vol. 68, pp. 456–461. North-Holland, Amsterdam (1969)
25. Ullman, J.: Principles of Knowledge and Database Systems, vol. 1. Computer Science Press (1988)
26. Van Hentenryck, P., Cortesi, A., Le Charlier, B.: Type analysis of Prolog using type graphs. Journal of Logic Programming 22(3), 179–210 (1994)
27. Vaucheret, C., Bueno, F.: More precise yet efficient type inference for logic programs. In: Hermenegildo, M.V., Puebla, G. (eds.) SAS 2002. LNCS, vol. 2477, pp. 102–116. Springer, Heidelberg (2002)
28. Whaley, J., Lam, M.S.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In: Pugh, W., Chambers, C. (eds.) PLDI, pp. 131–144. ACM, New York (2004)
29. Whaley, J., Unkel, C., Lam, M.S.: A bdd-based deductive database for program analysis (2004), http://bddbddb.sourceforge.net/

A Higher-Order Iterative Path Ordering

Cynthia Kop and Femke van Raamsdonk

Vrije Universiteit, Department of Theoretical Computer Science,
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands
kop@few.vu.nl, femke@few.vu.nl

Abstract. The higher-order recursive path ordering (HORPO) defined by Jouannaud and Rubio provides a method to prove termination of higher-order rewriting. We present an iterative version of HORPO by means of an auxiliary term rewriting system, following an approach originally due to Bergstra and Klop. We study well-foundedness of the iterative definition, discuss its relationship with the original HORPO, and point out possible ways to strengthen the ordering.

1 Introduction

This paper is about termination of higher-order rewriting, where bound variables may be present. An important method to prove termination of first-order term rewriting is provided by the recursive path ordering (RPO) defined by Dershowitz [4]. Jouannaud and Rubio [6] define the higher-order recursive path ordering (HORPO) which extends RPO to the higher-order case. The starting point is a well-founded ordering on the function symbols which is lifted to a relation $\succ$ on terms, such that if $l \succ^+ r$ for every rewrite rule $l \to r$, then rewriting terminates.

Klop, van Oostrom and de Vrijer [10] present, following an approach originally due to Bergstra and Klop [1], the iterative lexicographic path ordering (ILPO) by means of an auxiliary term rewriting system. ILPO can be understood as an iterative definition of the lexicographic path ordering (LPO), a variant of RPO [9]. They show that ILPO is well-founded, and that ILPO and LPO coincide if the underlying relation on function symbols is transitive.

The starting point of the present work is the question whether also for HORPO an iterative definition can be given. We present HOIPO, a higher-order iterative path ordering, which is defined by means of an auxiliary (higher-order) term rewriting system, following the approach of [10]. HOIPO can be considered as an extension of ILPO obtained by a generalization to the higher-order case and the addition of comparing arguments as multisets.

We show well-foundedness of HOIPO as in [6] using the notion of computability and the proof technique due to Buchholz [3], see also [5]. It then follows that HOIPO provides a method for proving termination of higher-order rewriting. Further, we show that HOIPO includes HORPO but not vice versa. So HOIPO is (slightly) stronger than HORPO as a termination method; the reason is that the fine-grained approach permits one to postpone the choice of smaller terms.

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 697–711, 2008.

2 Preliminaries

In this paper we mainly consider higher-order rewriting as defined by Jouannaud and Okada [5], also called Algebraic Functional Systems (AFSs) [16, Chapter 11]. The terms are simply typed λ-terms with typed constants. Every system has β as one of its rewrite rules. That is, in AFSs we do not work modulo β as for instance in HRSs [12]. Below we recall the main definitions.

Definition 1 (Types). We assume a set $\mathcal{S}$ of *sorts* also called *base types*. The set $\mathcal{T}$ of *simple types* is defined by the grammar $\mathcal{T} ::= \mathcal{S} \mid (\mathcal{T} \to \mathcal{T})$.

Types are denoted by $\sigma, \tau, \rho, \ldots$. As usual $\to$ associates to the right and we omit outermost parentheses. A *type declaration* is an expression of the form $(\sigma_1 \times \ldots \times \sigma_m) \to \sigma$ with $\sigma_1, \ldots, \sigma_m$ and σ types. If $m = 0$ then such a type declaration is shortly written as σ. Type declarations are not types, but are used for typing terms. In the remainder of this paper we assume a set $\mathcal{V}$ of typed variables, denoted by $x : \sigma, y : \tau, z : \rho, \ldots$, with countably many variables of every type. In the following definitions we assume in addition a set $\mathcal{F}$ of function symbols, each equipped with a unique type declaration, denoted by $f : (\sigma_1 \times \ldots \times \sigma_m) \to \sigma, g : (\tau_1 \times \ldots \times \tau_n) \to \tau, \ldots$.

Definition 2 (Terms). The set $\mathbb{T}(\mathcal{F}, \mathcal{V})$ of *terms* over $\mathcal{F}$ and $\mathcal{V}$ is the smallest set consisting of all expressions s for which we can infer $s : \sigma$ for some type σ using the following clauses:

(var)	$x : \sigma$	if $x : \sigma \in \mathcal{V}$
(app)	$@(u, t) : \sigma$	if $u : \tau \to \sigma$ and $t : \tau$
(abs)	$\lambda x : \tau. t : \tau \to \rho$	if $x : \tau \in \mathcal{V}$ and $t : \rho$
(fun)	$f(s_1, \ldots, s_m) : \sigma$	if $f : (\sigma_1 \times \ldots \times \sigma_m) \to \sigma \in \mathcal{F}$ and $s_1 : \sigma_1, \ldots, s_m : \sigma_m$

The application of n terms is sometimes written as $@(t_1, \ldots, t_n)$; here $n \geq 2$ and t_1 may be an application itself. Note that a function symbol $f : (\sigma_1 \times \ldots \times \sigma_m) \to \sigma$ must get exactly m arguments, and that σ is not necessarily a base type. Occurrences of x in t in the term $\lambda x : \tau. t$ are bound. We consider equality on terms modulo α-conversion, denoted by $=$. If we want to mention explicitly the type of a (sub)term then we write $s : \sigma$ instead of simply s.

A *substitution* $[\boldsymbol{x} := \boldsymbol{s}]$, with $\boldsymbol{x}$ and $\boldsymbol{s}$ finite vectors of equal length, is the homomorphic extension of the type-preserving mapping $\boldsymbol{x} \mapsto \boldsymbol{s}$ from variables to terms. Substitutions are denoted by $\gamma, \delta, \ldots$, and the result of applying the substitution γ to the term s is denoted by $s\gamma$. Substitutions do not capture free variables; we assume that bound variables are renamed if necessary.

Definition 3 (Rewrite rule). A *rewrite rule* over $\mathcal{F}$ and $\mathcal{V}$ is a pair of terms (l, r) in $\mathbb{T}(\mathcal{F}, \mathcal{V})$ of the same type, such that all free variables of r occur in l.

A rewrite rule (l, r) is usually written as $l \to r$. A *higher-order rewrite system* is specified by a set $\mathcal{F}$ of function symbols with type declarations, and a set of rewrite rules over $\mathcal{F}$ and $\mathcal{V}$. The rewrite rules induce a rewrite relation which is defined as follows; note that matching is modulo α, not modulo $\alpha\beta$ nor $\alpha\beta\eta$.

Definition 4 (Rewrite relations). Given a set of rewrite rules $\mathcal{R}$ over $\mathcal{F}$ and $\mathcal{V}$, the rewrite relation $\to_{\mathcal{R}}$ is defined by the following clauses:

(head)	$l\gamma \to_{\mathcal{R}} r\gamma$	if $l \to r \in \mathcal{R}$ and γ a substitution
(fun)	$f(s_1,\ldots,s_i,\ldots,s_n) \to_{\mathcal{R}} f(s_1,\ldots,s_i',\ldots,s_n)$	if $s_i \to_{\mathcal{R}} s_i'$
(app-l)	$@(s,t) \to_{\mathcal{R}} @(s',t)$	if $s \to_{\mathcal{R}} s'$
(app-r)	$@(s,t) \to_{\mathcal{R}} @(s,t')$	if $t \to_{\mathcal{R}} t'$
(abs)	$\lambda x:\sigma.\, s \to_{\mathcal{R}} \lambda x:\sigma.\, s'$	if $s \to_{\mathcal{R}} s'$

The *β-reduction relation*, denoted by $\to_\beta$, is induced by the β-reduction rule $@(\lambda x:\sigma.\, s,t) \to_\beta s[x := t]$. The *rewrite relation* of $(\mathcal{F},\mathcal{R})$, denoted by $\to$, is defined as the union of $\to_{\mathcal{R}}$ and β-reduction: $\to \;=\; \to_{\mathcal{R}} \cup \to_\beta$. As usual, the transitive closure of $\to$ is denoted by $\to^+$ and the reflexive-transitive closure of $\to$ is denoted by $\to^*$.

Example 1 (Recursor). The rewrite system Rec for recursor on natural numbers uses a base type $\mathbb{N}$ and function symbols $\mathsf{0} : \mathbb{N}$, $\mathsf{S} : (\mathbb{N}) \to \mathbb{N}$, $\mathsf{rec} : (\mathbb{N} \times \mathbb{N} \times (\mathbb{N} \to \mathbb{N} \to \mathbb{N})) \to \mathbb{N}$. The rewrite rules of Rec are as follows:

$$\begin{aligned} \mathsf{rec}(\mathsf{0},y,z) &\to y \\ \mathsf{rec}(\mathsf{S}(x),y,z) &\to @(z,x,\mathsf{rec}(x,y,z)) \end{aligned}$$

Addition of natural numbers can now be represented by the following term:

$$\lambda x:\mathbb{N}.\,\lambda y:\mathbb{N}.\,\mathsf{rec}(x,y,\lambda u:\mathbb{N}.\,\lambda v:\mathbb{N}.\,\mathsf{S}(v))$$

Example 2 (Map). The rewrite system Map uses base types $\mathbb{N}$ for natural numbers, and $\mathsf{natlist}$ for lists of natural numbers. The function symbols are $\mathsf{nil} : \mathsf{natlist}$, $\mathsf{cons} : (\mathbb{N} \times \mathsf{natlist}) \to \mathsf{natlist}$, $\mathsf{map} : (\mathsf{natlist} \times (\mathbb{N} \to \mathbb{N})) \to \mathsf{natlist}$. The rewrite rules of Map are as follows:

$$\begin{aligned} \mathsf{map}(\mathsf{nil},z) &\to \mathsf{nil} \\ \mathsf{map}(\mathsf{cons}(h,t),z) &\to \mathsf{cons}(@(z,h),\mathsf{map}(t,z)) \end{aligned}$$

3 The Higher-Order Recursive Path Ordering

The starting point of the recursive path ordering due to Dershowitz [4] is a well-founded ordering on the function symbols, which is lifted to a reduction ordering $\succ_{rpo}$ on the set of terms. That is, the rewriting system is terminating if $l \succ_{rpo} r$ for every rewrite rule $l \to r$. Jouannaud and Rubio [6] present an extension of RPO to the higher-order case, here called HORPO. Below we recall the definition of HORPO and in the next section we introduce its fine-grained iterative version.

We have chosen to work with the definition as in [6] and not with later versions as for instance [8,2]; we will come back to this issue in the last section.

In the following, multisets are denoted by $\{\!\{\ldots\}\!\}$. If $>$ is a binary relation, then the *multiset extension* of $>$, denoted by $>_{MUL}$, is defined as follows: $X \cup Y >_{MUL} X \cup Z$, with $\cup$ the disjoint union of multisets, if $Y \neq \emptyset$ and $\forall z \in Z.\, \exists y \in Y.\, y > z$.

Sequences are denoted by $[\ldots]$. The *lexicographic extension* of $>$, denoted by $>_{LEX}$, is defined as follows: $[s_1,\ldots,s_m] >_{LEX} [s'_1,\ldots,s'_m]$ if either $s_1 > s'_1$ or $s_1 = s'_1$ and $[s_2,\ldots,s_m] >_{LEX} [s'_2,\ldots,s'_m]$. If $>$ is a well-founded relation, then both $>_{MUL}$ and $>_{LEX}$ are well-founded.

We assume that the set of function symbols $\mathcal{F}$ is the disjoint union of $\mathcal{F}_{\mathsf{MUL}}$ and $\mathcal{F}_{\mathsf{LEX}}$. If $f \in \mathcal{F}_{\mathsf{MUL}}$ then its arguments will be compared with the multiset extension of HORPO, and if $f \in \mathcal{F}_{\mathsf{LEX}}$ then its arguments will be compared with the lexicographic extension of HORPO. We assume a well-founded precedence $\triangleright$ on $\mathcal{F}$. Finally, in the remainder we identify all base types.

Definition 5 (HORPO). We have $s \succ t$ for terms $s : \sigma$ and $t : \sigma$ if one of the following conditions holds:

(H1) $s = f(s_1,\ldots,s_m)$
there is an $i \in \{1,\ldots,m\}$ such that $s_i \succeq t$

(H2) $s = f(s_1,\ldots,s_m)$, $t = g(t_1,\ldots,t_n)$
$f \triangleright g$
$s \succ\!\!\succ \{t_1,\ldots,t_n\}$

(H3LEX) $s = f(s_1,\ldots,s_m)$, $t = f(t_1,\ldots,t_m)$, $f \in \mathcal{F}_{\mathsf{LEX}}$
$[s_1,\ldots,s_m] \succ_{LEX} [t_1,\ldots,t_m]$
$s \succ\!\!\succ \{t_1,\ldots,t_m\}$

(H3MUL) $s = f(s_1,\ldots,s_m)$, $t = f(t_1,\ldots,t_m)$, $f \in \mathcal{F}_{\mathsf{MUL}}$
$\{\{s_1,\ldots,s_m\}\} \succ_{MUL} \{\{t_1,\ldots,t_m\}\}$

(H4) $s = f(s_1,\ldots,s_m)$, $t = @(t_1,\ldots,t_n)$ with $n \geq 2$
$s \succ\!\!\succ \{t_1,\ldots,t_n\}$

(H5) $s = @(s_1,s_2)$, $t = @(t_1,t_2)$
$\{\{s_1,s_2\}\} \succ_{MUL} \{\{t_1,t_2\}\}$

(H6) $s = \lambda x:\sigma.\, s_0$, $t = \lambda x:\sigma.\, t_0$
$s_0 \succ t_0$

Here $\succeq$ denotes the reflexive closure of $\succ$, and $\succ_{MUL}$ and $\succ_{LEX}$ denote the multiset and lexicographic extension of $\succ$. Further, following the notation from [11], the relation $\succ\!\!\succ$ between a functional term and a set of terms is defined as follows: $s = f(s_1,\ldots,s_m) \succ\!\!\succ \{t_1,\ldots,t_n\}$ if for every $i \in \{1,\ldots,n\}$ we have either $s \succ t_i$, or there exists $j \in \{1,\ldots,m\}$ such that $s_j \succeq t_i$.

The first four clauses of the definition stem directly from the first-order definition of RPO, with the difference that instead of the requirement $s \succ\!\!\succ \{t_1,\ldots,t_n\}$ for HORPO, we have for RPO the simpler $s \succ t_i$ for every i. This is not possible for the higher-order case because of the type requirements; the relation $\succ$ is only defined on terms of equal type (after identifying all base types).

Jouannaud and Rubio prove well-foundedness of $\succ \cup \to_\beta$. HORPO provides a method to prove termination of higher-order rewriting: a higher-order rewrite system $(\mathcal{F},\mathcal{R})$ is terminating if $l \succ r$ for every rewrite rule $l \to r \in \mathcal{R}$.

Example 3 (Recursor). We prove termination of the recursor on natural numbers using HORPO. For the first rewrite rule, we have $\mathsf{rec}(0,y,z) \succ y$ by (H1). We use (H4) to show $\mathsf{rec}(\mathsf{S}(x),y,z) \succ @(z,x,\mathsf{rec}(x,y,z))$. There are three remaining

proof obligations. First, we have $z \succeq z$ by reflexivity of $\succeq$. Second, we have $\mathsf{S}(x) \succeq x$ by clause (H1). Third, we have $\mathsf{rec}(\mathsf{S}(x), y, z) \succ \mathsf{rec}(x, y, z)$ by assuming rec to be a lexicographic function symbol and using again $\mathsf{S}(x) \succ x$, and reflexivity.

Example 4 (Map). The first rewrite rule is oriented using (H1). In order to orient the second rewrite rule we apply (H2) with $\mathsf{map} \triangleright \mathsf{cons}$. Then first we need to show $\mathsf{map}(\mathsf{cons}(h,t),z) \succ @(z,h)$ which follows from (H4). Second we need to show $\mathsf{map}(\mathsf{cons}(h,t),z) \succ \mathsf{map}(t,z)$ using (H3MUL). (Alternatively one can assume map to be a lexicographic function symbol.) Note that in this example we need the collapse of all base types to one.

4 An Iterative Version of HORPO

In this section we present an iterative version of HORPO, called HOIPO. HOIPO is defined by means of a term rewriting system that intuitively step by step transforms a term into a term that is smaller with respect to HORPO. We will add marked function symbols: if $\mathcal{F}$ is a set of function symbols, then $\mathcal{F}^*$ is defined to be a copy of $\mathcal{F}$ which contains for every $f \in \mathcal{F}$ a symbol f^* with the same type declaration. We follow the approach and notations as in [10].

Definition 6 (HOIPO). We assume a set of function symbols $\mathcal{F}$ divided in $\mathcal{F}_{\mathsf{MUL}}$ and $\mathcal{F}_{\mathsf{LEX}}$, and a relation $\triangleright$ on $\mathcal{F}$. The rewriting system $\mathcal{H}(\mathcal{F}, \triangleright)$ uses function symbols in $\mathcal{F} \cup \mathcal{F}^*$ and contains the following rules:

$$
\begin{array}{rll}
f(x_1,\ldots,x_m) & \to_{\text{put}} & f^*(x_1,\ldots,x_m) \\
f^*(x_1,\ldots,x_m) & \to_{\text{select}} & x_i \qquad \text{(a)} \\
f^*(x_1,\ldots,x_m) & \to_{\text{copy}} & g(l_{\tau_1},\ldots,l_{\tau_n}) \qquad \text{(b)(f)} \\
f^*(x_1,\ldots,s_i,\ldots,x_m) & \to_{\text{lex}} & f(x_1,\ldots,x_{i-1},s_i',l_{\sigma_{i+1}},\ldots,l_{\sigma_m}) \qquad \text{(c)(d)(f)} \\
f^*(x_1,\ldots,s_i,\ldots,x_m) & \to_{\text{mul}} & f(r_1,\ldots,r_{i-1},s_i',r_{i+1},\ldots,r_m) \qquad \text{(c)(e)(g)} \\
f^*(x_1,\ldots,x_m) & \to_{\text{ord}} & f^*(x_{\pi^{-1}1},\ldots,x_{\pi^{-1}m}) \qquad \text{(e)(h)} \\
f^*(x_1,\ldots,x_m) & \to_{\text{appl}} & @(l_{\rho_1\to\ldots\to\rho_k\to\sigma},l_{\rho_1},\ldots,l_{\rho_k}) \qquad \text{(f)}
\end{array}
$$

We assume that the typing and arity constraints are met (after identifying all base types). In particular the left- and right-hand sides of rewrite rules must have the same type, and $f : (\sigma_1 \times \ldots \times \sigma_m) \to \sigma$, and $g : (\tau_1 \times \ldots \times \tau_n) \to \sigma$. Further, the rules are subject to the following conditions:

(a) $i \in \{1,\ldots,m\}$,
(b) $f \triangleright g$,
(c) $s_i \to_{\text{put}} s_i'$,
(d) $f \in \mathcal{F}_{\mathsf{LEX}}$,
(e) $f \in \mathcal{F}_{\mathsf{MUL}}$,
(f) we use the notation l_ρ for some term of type ρ; either $l_\rho = f^*(x_1,\ldots,x_m)$ or $l_\rho = x_j$ for some j, as long as the type constraints are met; it is not a fixed term: we can choose different values for l_{σ_i} and l_{σ_j} even if $\sigma_i = \sigma_j$,
(g) we use the notation r_j for some term of type σ_j: either $s_i \to_{\text{put}} r_j$, or $r_j = x_j$
(h) π a type-preserving permutation of $1,\ldots,m$.

The first four rewrite rules stem directly from the first-order term rewriting system $\mathcal{L}\mathsf{ex}$ defined in [10] which is used to define ILPO, an iterative definition of the lexicographic path order. They are adapted because of the typing constraints, just as the clauses (H1), (H2), (H3MUL), (H3LEX) in the definition of HORPO are typed versions of the clauses of the definition of first-order RPO. We now first discuss the intuitive meaning of the rewrite rules of HOIPO and then give some examples.

- The put-rule can be considered as the start of a proof obligation for HORPO. It expresses the intention to make a functional term smaller. It is exactly the same as the put-rule of the first-order rewriting system for ILPO.
- The select-rule expresses that selecting a direct argument of a functional term makes it smaller. It roughly corresponds to clause (H1) of the definition of HORPO. It is exactly the same as the select-rule in the rewriting system for ILPO. However, the use of the rule in the higher-order setting is weaker because of the typing constraints. For instance, with $f : (o \to o) \to o$, $g : o \to (o \to o)$, $a : o$, we cannot reduce $f(g(a)) : o$ to $a : o$ in $\mathcal{H}$ using the rules put and select, because we would need to go via $g(a)$ which has type $o \to o$. In the first-order setting we have $f(g(a)) \to_{\text{put}} f^*(g(a)) \to_{\text{select}} g(a) \to_{\text{put}} g^*(a) \to_{\text{select}} a$.
- The copy-rule makes copies of the original term under a function symbol that is smaller with respect to $\rhd$. This corresponds to clause (H2) of the definition of HORPO. The choice for l_{σ_i} in the right-hand side of the rule corresponds to the $\succ\succ$ relation used in (H2). The first-order version of the rule is
$$f^*(\boldsymbol{x}) \to_{\text{copy}} g(f^*(\boldsymbol{x}), \ldots, f^*(\boldsymbol{x})) \qquad (\text{if } f \rhd g)$$
There the left-hand side is copied at all argument positions of g, which cannot be done in the higher-order case because of the typing constraints.
- The lex-rule implements the lexicographic extension of HORPO and can be applied to lexicographic functional terms only. The first $i-1$ arguments are not changed. The ith argument is marked, meaning we will make it smaller. The arguments $i+1$ till m may increase, but are bounded by the left-hand side. That is, on those positions we put either the original term (left-hand side) or a direct argument. The first-order version of this rewrite rule is:
$$f^*(\boldsymbol{x}, g(\boldsymbol{y}), \boldsymbol{z}) \to_{\text{lex}} f(\boldsymbol{x}, g^*(\boldsymbol{y}), l, \ldots, l) \qquad (\text{for } l = f^*(\boldsymbol{x}, g(\boldsymbol{y}), \boldsymbol{z}))$$
Here the put-reduct of the argument $g(\boldsymbol{y})$ can be given directly, and at all positions thereafter the left-hand side can be put. For the higher-order case this is not possible because instead of a functional term we can also have an application or an abstraction.
- The mul- and ord-rule implement the multiset extension of HORPO and can be applied to multiset functional terms only. With $a \rhd b$ we have for instance $f(x,a,c) \to_{\text{put}} f^*(x,a,c) \to_{\text{ord}} f^*(x,c,a) \to_{\text{mul}} f(a^*,c,a^*) \to_{\text{copy}} f(b,c,a^*) \to_{\text{copy}} f(b,c,b)$. We cannot reduce $f(x,a,c)$ to $f(b,c,b)$ using the first-order rules put, select, copy, and lex, so the mul-rule cannot be derived from those rules.

The ord-rule expresses that the order of the arguments of a multiset functional term do not matter (but remain subject to the typing constraints). This rule does not express a decrease in HORPO.

– The appl-rule corresponds to clause (H4) of the definition of HORPO and is typical for the higher-order case. The idea is that the application of terms that are all smaller than the original term is smaller than the original term. We have $l_{\rho_1\to\ldots\to\rho_k\to\sigma} = x_i$ and $\rho_1 \to \ldots \to \rho_k \to \sigma = \sigma_i$ for some i.

The rewrite relation induced by the rules of $\mathcal{H}(\mathcal{F}, \rhd)$ is denoted by $\to_{\mathcal{H}}$, and the union of $\to_{\mathcal{H}}$ and β-reduction is denoted by $\to_{\mathcal{H}\beta}$. How can HOIPO be used to prove termination? The claim is that a system $(\mathcal{F}, \mathcal{R})$ is terminating if we have $l \to^+_{\mathcal{H}\beta} r$ for every rewrite rule $l \to r \in \mathcal{R}$. This is proved in the following section; here we first look at three examples of the use of HOIPO.

Example 5 (Recursor). For the first rewrite rule we have:

$$\mathsf{rec}(0, y, z) \to_{\mathrm{put}} \mathsf{rec}^*(0, y, z) \to_{\mathrm{select}} y$$

For the second rewrite rule we assume $\mathsf{rec} \in \mathcal{F}_{\mathsf{LEX}}$. Then:

$$\begin{array}{l}\mathsf{rec}(\mathsf{S}(x), y, z) \to_{\mathrm{put}} \mathsf{rec}^*(\mathsf{S}(x), y, z) \to_{\mathrm{appl}} @(z, \mathsf{S}(x), \mathsf{rec}^*(\mathsf{S}(x), y, z)) \to_{\mathrm{put}} \\ @(z, \mathsf{S}^*(x), \mathsf{rec}^*(\mathsf{S}(x), y, z)) \to_{\mathrm{select}} @(z, x, \mathsf{rec}^*(\mathsf{S}(x), y, z)) \to_{\mathrm{lex}} \\ @(z, x, \mathsf{rec}(\mathsf{S}^*(x), y, z)) \to_{\mathrm{select}} @(z, x, \mathsf{rec}(x, y, z))\end{array}$$

Example 6 (Map). We take $\mathsf{map} \rhd \mathsf{cons}$ and $\mathsf{map} \in \mathcal{F}_{\mathsf{LEX}}$. For the first rewrite rule we have:

$$\mathsf{map}(\mathsf{nil}, z) \to_{\mathrm{put}} \mathsf{map}^*(\mathsf{nil}, z) \to_{\mathrm{select}} \mathsf{nil}$$

For the second rewrite rule we have (base types are identified):

$$\begin{array}{l}\mathsf{map}(\mathsf{cons}(h, t), z) \to_{\mathrm{put}} \mathsf{map}^*(\mathsf{cons}(h, t), z) \to_{\mathrm{copy}} \\ \mathsf{cons}(\mathsf{map}^*(\mathsf{cons}(h, t), z), \mathsf{map}^*(\mathsf{cons}(h, t), z)) \to_{\mathrm{appl}} \\ \mathsf{cons}(@(z, \mathsf{cons}(h, t)), \mathsf{map}^*(\mathsf{cons}(h, t), z)) \to_{\mathrm{put}} \\ \mathsf{cons}(@(z, \mathsf{cons}^*(h, t)), \mathsf{map}^*(\mathsf{cons}(h, t), z)) \to_{\mathrm{select}} \\ \mathsf{cons}(@(z, h), \mathsf{map}^*(\mathsf{cons}(h, t), z)) \to_{\mathrm{lex}} \mathsf{cons}(@(z, h), \mathsf{map}(\mathsf{cons}^*(h, t), z)) \to_{\mathrm{select}} \\ \mathsf{cons}(@(z, h), \mathsf{map}(t, z))\end{array}$$

5 Termination

In this section we prove the following theorem:

Theorem 1. *A system $(\mathcal{F}, \mathcal{R})$ is terminating if there exists a well-founded ordering $\rhd$ on the terms over $\mathcal{F}$ such that $l \to^+_{\mathcal{H}\beta} r$ in $\mathcal{H}(\mathcal{F}, \rhd)$.*

This means that HOIPO provides a termination method, as was already claimed in the previous section. The proof of Theorem 1 proceeds as follows; we follow the approaches of [10,6]. We define a labelled rewrite relation $\to_{\mathcal{H}\omega}$ (Definition 7) and consider its union with β-reduction, denoted by $\to_{\mathcal{H}\omega\beta}$. It is shown that

$\to^+_{\mathcal{H}\omega\beta}$ and $\to^+_{\mathcal{H}\beta}$ coincide on the set of terms over $\mathcal{F}$, so without marks or labels (Lemma 1). Then we show termination of $\to_{\mathcal{H}\omega\beta}$ (Theorem 3). It follows that $\to^+_{\mathcal{H}\beta}$ is a transitive relation on the terms over $\mathcal{F}$, that is closed under contexts and substitutions, and that is moreover well-founded. Hence it is a reduction ordering, that is, $l \to^+_{\mathcal{H}\beta} r$ for every rewrite rule $l \to r$ implies that rewriting is terminating. HOIPO is more fine-grained than HORPO and also its termination proof is more fine-grained that the one for HORPO. We continue by presenting the labelled version of HOIPO, which is used to prove termination of $\to_{\mathcal{H}\beta}$.

5.1 The Labelled System

We assume a set $\mathcal{F}$ of function symbols, which is the disjoint union of lexicographic and multiset function symbols, and a well-founded ordering $\rhd$ on $\mathcal{F}$. We define a copy $\mathcal{F}^\omega$ of $\mathcal{F}$ as follows: for every $f \in \mathcal{F}$, the set $\mathcal{F}^\omega$ contains the labelled function symbol f^n for every natural number n. An unlabelled function symbol f is also denoted as f^ω; then every function symbol in $\mathcal{F} \cup \mathcal{F}^\omega$ can be denoted by f^α with α an ordinal of at most ω. The usual ordering on $\mathbb{N}$ is extended by $n < \omega$ for every $n \in \mathbb{N}$.

Definition 7 (Labelled HOIPO). The rewriting system $\mathcal{H}\omega(\mathcal{F}, \rhd)$ uses function symbols in $\mathcal{F} \cup \mathcal{F}^\omega$ and contains the following rules:

$$
\begin{array}{rll}
f^\omega(x_1,\dots,x_m) & \to_{\text{put}} & f^p(x_1,\dots,x_m) \\
f^p(x_1,\dots,x_m) & \to_{\text{select}} & x_i \qquad \text{(a)} \\
f^{p+1}(x_1,\dots,x_m) & \to_{\text{copy}} & g^\omega(l_{\tau_1},\dots,l_{\tau_n}) \qquad \text{(b)(f)} \\
f^{p+1}(x_1,\dots,s_i,\dots,x_m) & \to_{\text{lex}} & f^\omega(x_1,\dots,x_{i-1},s_i',l_{\sigma_{i+1}},\dots,l_{\sigma_m}) \qquad \text{(c)(d)(f)} \\
f^{p+1}(x_1,\dots,s_i,\dots,x_m) & \to_{\text{mul}} & f^\omega(r_1,\dots,r_{i-1},s_i',r_{i+1},\dots,r_m) \qquad \text{(c)(e)(g)} \\
f^{p+1}(x_1,\dots,x_m) & \to_{\text{ord}} & f^p(x_{\pi^{-1}1},\dots,x_{\pi^{-1}m}) \qquad \text{(e)(h)} \\
f^{p+1}(x_1,\dots,x_m) & \to_{\text{appl}} & @(l_{\rho_1\to\dots\to\rho_k\to\sigma},l_{\rho_1},\dots,l_{\rho_k}) \qquad \text{(f)}
\end{array}
$$

Here p is an arbitrary natural number. As in Definition 6 we assume that the type- and arity constraints are met. In addition we have the following conditions; here (a), (b), (d), (e), and (h) are exactly the same as in Definition 6, and (c), (f) and (g), provide the crucial differences.

(a) $i \in \{1,\dots,m\}$,
(b) $f \rhd g$,
(c) $s_i \to_{\text{put}} s_i'$ with label p
(d) $f \in \mathcal{F}_{\mathsf{LEX}}$,
(e) $f \in \mathcal{F}_{\mathsf{MUL}}$,
(f) we use the notation l_ρ for some term of type ρ; either $l_\rho = f^p(x_1,\dots,x_m)$ or $l_\rho = x_j$ for some j, as long as the type constraints are met; it is not a fixed term: we can choose different values for l_{σ_i} and l_{σ_j} even if $\sigma_i = \sigma_j$,
(g) we use the notation r_j for some term of type σ_j: either $s_i \to_{\text{put}} r_j$ with label p, or $r_j = x_j$,
(h) π a type-preserving permutation of $1,\dots,m$.

The rewrite relation induced by the rewrite rules of $\mathcal{H}\omega(\mathcal{F}, \rhd)$ is denoted by $\to_{\mathcal{H}\omega}$, and the union of $\to_{\mathcal{H}\omega}$ and β-reduction is denoted by $\to_{\mathcal{H}\omega\beta}$. The following examples illustrate that in $\mathcal{H}\omega$ the labels provide more control over the reduction relation than the marks.

Example 7 (Map). We take $\mathsf{map} \rhd \mathsf{cons}$ and $\mathsf{map} \in \mathcal{F}_{\mathsf{LEX}}$. For the first rewrite rule we have:

$$\mathsf{map}(\mathsf{nil}, z) \to_{\mathrm{put}} \mathsf{map}^0(\mathsf{nil}, z) \to_{\mathrm{select}} \mathsf{nil}$$

For the second rewrite rule we have:

$$\begin{array}{l}\mathsf{map}(\mathsf{cons}(h,t),z) \to_{\mathrm{put}} \mathsf{map}^2(\mathsf{cons}(h,t),z) \to_{\mathrm{copy}}\\ \mathsf{cons}(\mathsf{map}^1(\mathsf{cons}(h,t),z), \mathsf{map}^1(\mathsf{cons}(h,t),z)) \to_{\mathrm{appl}}\\ \mathsf{cons}(@(z,\mathsf{cons}(h,t)), \mathsf{map}^1(\mathsf{cons}(h,t),z)) \to_{\mathrm{put}}\\ \mathsf{cons}(@(z,\mathsf{cons}^0(h,t)), \mathsf{map}^1(\mathsf{cons}(h,t),z)) \to_{\mathrm{select}}\\ \mathsf{cons}(@(z,h), \mathsf{map}^1(\mathsf{cons}(h,t),z)) \to_{\mathrm{lex}} \mathsf{cons}(@(z,h), \mathsf{map}(\mathsf{cons}^0(h,t),z)) \to_{\mathrm{select}}\\ \mathsf{cons}(@(z,h), \mathsf{map}(t,z))\end{array}$$

Example 8. As remarked in [10], the rewriting system $\to_{\mathcal{H}}$ may contain loops. For instance, for $a : \sigma$ and $f : \sigma \to \sigma$ with $a \rhd f$ we have $a \to_{\mathrm{put}} a^* \to_{\mathrm{copy}} f(a^*) \to_{\mathrm{put}} f^*(a^*) \to_{\mathrm{select}} a^*$. The loop on marked terms has no counterpart in the labelled system.

We now show that the reflexive closures $\to^+_{\mathcal{H}\beta}$ and $\to^+_{\mathcal{H}\omega\beta}$ coincide on the set of terms without labels.

Lemma 1. *We have $\to^+_{\mathcal{H}\beta} = \to^+_{\mathcal{H}\omega\beta}$ on the set of terms over $\mathcal{F}$.*

Proof. First note that the marks and labels do not control β-reduction.

In order to show $\to^+_{\mathcal{H}\beta} \subseteq \to^+_{\mathcal{H}\omega\beta}$ we consider a rewrite sequence $s \to^+_{\mathcal{H}\beta} t$ that does not consist of β-reduction steps only. The first step with respect to HOIPO is induced by the put-rule. The total number of HOIPO steps is finite, say n. We now lift the rewrite sequence in $\mathcal{H}\beta$ to a rewrite sequence in $\mathcal{H}\omega\beta$ by using in the first put-step of the latter the label n. The values for the other labels then follow easily.

In order to show $\to^+_{\mathcal{H}\omega\beta} \subseteq \to^+_{\mathcal{H}\beta}$, note that a step in labelled HOIPO is mapped to a step in HOIPO by replacing a label p (for any natural number p) by the mark $*$. A label ω is just a notational device and in fact denotes 'no label'. □

5.2 Computability

We will make use of the notion of computability due to Tait and Girard [15]. Here we define computability with respect to $\to_{\mathcal{H}\omega\beta}$.

Definition 8. A term $s : \sigma$ is said to be *computable* with respect to $\to_{\mathcal{H}\omega\beta}$ if:

- σ is a base type, and s is strongly normalizing with respect to $\to_{\mathcal{H}\omega\beta}$,
- $\sigma = \tau \to \rho$ for some τ and ρ, and for every $t : \tau$ with t computable with respect to $\to_{\mathcal{H}\omega\beta}$ we have that $@(s,t)$ is computable with respect to $\to_{\mathcal{H}\omega\beta}$.

As in [6,11] variables are not by definition computable, but are proved to be computable. We do not consider computability modulo a convertibility relation on terms as in [11] because we work with typed variables and not with environments. The following two lemma's are concerned with (standard) properties of computability and correspond to Property 3.4 (i), (ii), (iii), (v) in [6] and to Lemma 6.3 and Lemma 6.5 in [11].

Lemma 2

(a) If $s : \sigma$ is computable then $s : \sigma$ is strongly normalizing with respect to $\rightarrow_{\mathcal{H}\omega\beta}$.
(b) If $s : \sigma$ is computable and $s \rightarrow_{\mathcal{H}\omega\beta} t$ then t is computable.
(c) If $s : \sigma$ is not an abstraction (or: $s : \sigma$ is neutral) and t is computable for every t with $s \rightarrow_{\mathcal{H}\omega\beta} t$, then s is computable.

The three items are proved simultaneously by induction on the structure of type. A consequence of the third point is that variables are computable.

Lemma 3. *Consider an abstraction $\lambda x : \sigma.\, s : \sigma \rightarrow \tau$. If $s[x := t]$ is computable for every computable $t : \sigma$, then $\lambda x : \sigma.\, s$ is computable.*

The proof proceeds by showing that all one-step reducts of $@(\lambda x : \sigma.\, s, t)$ are computable and then applying Lemma 2(c).

We proceed by showing that a functional term $f(s_1, \ldots, s_m)$ is computable if all its direct arguments are computable. The proof of the following lemma employs a technique due to Buchholz [3], also already present in [5], that is for instance also used in [6,10].

Lemma 4. *If $s_1 : \sigma_1, \ldots, s_m : \sigma_m$ are computable, then $f^\alpha(s_1, \ldots, s_m)$ is computable.*

Proof. Assume computable terms $s_1 : \sigma_1, \ldots, s_m : \sigma_m$ and a function symbol $f : (\sigma_1 \times \ldots \times \sigma_m) \rightarrow \tau$, and an ordinal α with $\alpha \leq \omega$. We prove that $f^\alpha(s_1, \ldots, s_m)$ is computable by well-founded induction on the triple $(f, \boldsymbol{s}, \alpha)$, ordered by the lexicographic product of the following three orderings: first $\rhd$ on function symbols, second the lexicographic (for $f \in \mathcal{F}_{\mathsf{LEX}}$) or multiset (for $f \in \mathcal{F}_{\mathsf{MUL}}$) extension of $\rightarrow_{\mathcal{H}\omega\beta}$ on vectors of computable terms, and third the ordering $>$ on natural numbers extended with $\omega > n$ for every $n \in \mathbb{N}$. We denote this ordering by $>>$.

The induction hypothesis is: If $(f, \boldsymbol{s}, \alpha) >> (g, \boldsymbol{t}, \beta)$ with $\boldsymbol{t} = t_1, \ldots, t_n$ computable terms, then $g^\beta(t_1, \ldots, t_n)$ is computable.

Because $s = f^\alpha(s_1, \ldots, s_m)$ is neutral (i.e. not an abstraction), by Lemma 2(c) it is sufficient to prove that all one-step reducts of s are computable. So we suppose that $f^\alpha(s_1, \ldots, s_m) \rightarrow_{\mathcal{H}\omega\beta} t$ and proceed by showing computability of t. If the rewrite step takes place inside one of the s_i, then t is computable by the induction hypothesis for the second component (note that f does not increase). Otherwise, $f^\alpha(s_1, \ldots, s_k) \rightarrow_{\mathcal{H}\omega\beta} t$ is a head step. We consider 4 of the 7 possibilities.

- Suppose that $f^\omega(s_1,\dots,s_m) \to_{\mathrm{put}} f^p(s_1,\dots,s_m)$. (Note that in this case $\alpha = \omega$.) Then $t = f^p(s_1,\dots,s_m)$ is computable by the induction hypothesis for the third component, because the first two components of the triple do not change, and $\omega > p$ for every natural number p.
- Suppose that $f^{p+1}(s_1,\dots,s_m) \to_{\mathrm{copy}} g^\omega(t_1,\dots,t_n)$. For every t_j with $j \in \{1,\dots,n\}$ we have either $t_j = f^p(s_1,\dots,s_m)$ or $t_j = s_k$ for some $k \in \{1,\dots,m\}$. In the first case t_j is computable by the induction hypothesis on the third component. In the second case t_j is computable by assumption. Therefore $(t_1,\dots,t_n)$ consists of computable terms. Now computability of $t = g(t_1,\dots,t_n)$ follows by the induction hypothesis on the first component.
- Suppose that $f^{p+1}(s_1,\dots,s_m) \to_{\mathrm{mul}} f^\omega(t_1,\dots,t_{i-1},s'_i,t_{i+1},\dots,t_m)$. For every $j \in \{1,\dots,i-1,i+1,\dots,m\}$ we have either $s_i \to_{\mathrm{put}} t_j$ or $t_j = s_j$. In the first case t_j is computable by the assumption that s_i is computable and Lemma 2(b). In the second case t_j is computable by assumption. Further, s'_i is a put-reduct of s_i and hence computable by assumption and Lemma 2(b). We conclude that $(t_1,\dots,t_{i-1},s'_i,t_{i+1},\dots,t_m)$ consists of computable terms. Now computability of $t = f^\omega(t_1,\dots,t_{i-1},s'_i,t_{i+1},\dots,t_m)$ follows by the induction hypothesis on the second component, because the multiset $\{\{s_1,\dots,s_m\}\}$ is greater than the multiset $\{\{t_1,\dots,t_{i-1},s'_i,t_{i+1},\dots,t_m\}\}$ in the multiset extension of $\to_{\mathcal{H}\omega\beta}$.
- Suppose that $f^{p+1}(s_1,\dots,s_m) \to_{\mathrm{appl}} @(t_0,t_1,\dots,t_n)$. For every $j \in \{0,\dots,n\}$ we have either $t_j = f^p(s_1,\dots,s_m)$ or $t_j = s_k$ for some $k \in \{1,\dots,m\}$. In the first case, t_j is computable by the induction hypothesis on the third component of the triple. In the second case, t_j is computable by assumption. Now computability of t follows because by definition the application of computable terms is computable.

From the complete case analysis follows that all one-step reducts of s are computable. Hence by Lemma 2(c) the term s is computable. □

We now show that all terms (possibly with labels) are computable, by showing the stronger statement that the application of a computable substitution to an arbitrary term yields a computable term. A substitution is said to be *computable* if all terms in its range are computable. The proof of the following theorem proceeds by induction on the definition of terms.

Theorem 2 (Computability of all terms). *Let $s : \sigma$ be an arbitrary term over $\mathcal{F} \cup \mathcal{F}^\omega$ and let γ be a computable substitution. Then $s\gamma$ is computable.*

Because the empty substitution is computable, it follows from this theorem that all terms over $\mathcal{F} \cup \mathcal{F}^\omega$ are computable. By Lemma 2 it then follows that all terms are strongly normalizing with respect to $\to_{\mathcal{H}\omega\beta}$.

Theorem 3 (Termination). *The rewriting relation $\to_{\mathcal{H}\omega\beta}$ is terminating on the set of terms over $\mathcal{F} \cup \mathcal{F}^\omega$.*

This concludes the proof of Theorem 1.

6 HOIPO Contains HORPO

The rewriting system $\mathcal{H}$ follows the definition of HORPO quite closely. In this section we show that indeed HOIPO contains HORPO. We assume a set of function symbols $\mathcal{F}$ and work also with marked terms over $\mathcal{F} \cup \mathcal{F}^*$.

Theorem 4. *Let s and t be terms over $\mathcal{F}$. If $s \succ t$ then $s \to_{\mathcal{H}}^* t$.*

Proof. Assume $s \succ t$. We prove by induction on the structure of s and t that there is some s' such that $s \to_{\text{put}} s' \to_{\mathcal{H}}^* t$. The induction hypothesis (IH) is: for all q and r, if q is a subterm of s, or ($q = s$ and r is a subterm of t), then $q \succ r$ implies $q \to_{\text{put}} q' \to_{\mathcal{H}}^* r$ (for some term q'). We consider all possible cases for $s \succ t$.

(H1) We have $s = f(s_1, \ldots, s_m)$ and there is an $i \in \{1, \ldots, m\}$ such that $s_i \succeq t$. If $s_i \succ t$ then by the IH (first component) $s_i \to_{\text{put}} s_i' \to_{\mathcal{H}}^* t$. If $s_i = t$ then also $s_i \to_{\mathcal{H}}^* t$. Hence in both cases $s \to_{\text{put}} f^*(s_1, \ldots, s_m) \to_{\text{select}} s_i \to_{\mathcal{H}}^* t$.

(H2) We have $s = f(s_1, \ldots, s_m)$, $t = g(t_1, \ldots, t_n)$, $f \rhd g$, and $s \succ\!\!\succ \{t_1, \ldots, t_n\}$.

For every $i \in \{1, \ldots, n\}$ we have either $s \succ t_i$ or $s_j \succeq t_i$ for some $j \in \{1, \ldots, m\}$. In the first case we define $l_i = f^*(s_1, \ldots, s_m)$. In the second case we define $l_i = s_j$.

In the first case we have by the IH $s \to_{\text{put}} s' \to_{\mathcal{H}}^* t_i$. Because for this s' (which is a put-reduct of s) either $s' = f^*(s_1, \ldots, s_m)$ or $f^*(s_1, \ldots, s_m) \to_{\text{lex/mul}} s'$, this yields $l_i = f^*(s_1, \ldots, s_m) \to_{\mathcal{H}}^* t_i$.

In the second case, if $s_j \succ t_i$ for some $j \in \{1, \ldots, m\}$, we have by the IH $s_j \to_{\text{put}} s_j' \to_{\mathcal{H}}^* t_i$. Hence we have $l_i = s_j \to_{\mathcal{H}}^* t_i$ (also if $s_j = t_i$).

Now we have $s \to_{\text{put}} f^*(s_1, \ldots, s_m) \to_{\text{copy}} g(l_1, \ldots, l_n) \to_{\mathcal{H}}^* g(t_1, \ldots, t_n)$.

(H3LEX) We have $s = f(s_1, \ldots, s_m)$ and $t = f(t_1, \ldots, t_m)$ with $f \in \mathcal{F}_{\mathsf{LEX}}$, and $[s_1, \ldots, s_m] \succ_{LEX} [t_1, \ldots, t_m]$ and $s \succ\!\!\succ \{t_1, \ldots, t_m\}$.

There is an $i \in \{1, \ldots, m\}$ such that $s_1 = t_1, \ldots, s_{i-1} = t_{i-1}, s_i \succ t_i$. Moreover, for every $j \in \{i+1, \ldots, m\}$ we have either $s \succ t_j$ or $s_k \succeq t_j$ for some k. By an analysis as in (H2) we can define for every $j \in \{i+1, \ldots, m\}$ a term l_j such that $l_j \to_{\mathcal{H}}^* t_j$. Because $s_i \succ t_i$ we have by the IH some term s_i' such that $s_i \to_{\text{put}} s_i' \to_{\mathcal{H}}^* t_i$. Combining this yields $s \to_{\text{put}} f^*(s_1, \ldots, s_m) \to_{\text{lex}} f(s_1, \ldots, s_{i-1}, s_i', l_{i+1}, \ldots, l_n) \to_{\mathcal{H}}^* f(s_1, \ldots, s_{i-1}, t_i, t_{i+1}, \ldots, t_n)$.

(H3MUL) We have $s = f(s_1, \ldots, s_m)$, $t = f(t_1, \ldots, t_m)$, $f \in \mathcal{F}_{\mathsf{MUL}}$, and moreover $\{\{s_1, \ldots, s_m\}\} \succ_{MUL} \{\{t_1, \ldots, t_m\}\}$.

By definition of the multiset ordering, we can write $\{\{s_1, \ldots, s_m\}\} = A \cup B \cup C$, $\{\{t_1, \ldots, t_m\}\} = A \cup D$, for all $t_i \in D$ there is some $s_j \in B$ such that $s_j \succ t_i$, and, since we are working with multisets of equal size, $|B| \geq 1$.

Suppose $|B| = 1$; write $A = \{\{s_{i_1}, \ldots, s_{i_k}\}\} = \{\{t_{j_1}, \ldots, t_{j_k}\}\}$, $B = \{\{s_n\}\}$. Since $s_n > t_x$ for all $x \notin \{j_1, \ldots, j_k\}$ we can derive by the induction hypothesis that for such x there is some s_n' where $s_n \to_{\text{put}} s_n' \to_{\mathcal{H}}^* t_x$.

Now, let π be a permutation that maps each i_x to j_x; then always $s_{\pi^{-1}j_x} = t_{j_x}$, and $s_{\pi^{-1}(\pi n)} > t_{\pi n}$ (because πn is not one of the j_x). So $f(s_1,\ldots,s_m) \to_{\text{put}} f^*(s_1,\ldots,s_m) \to_{\text{ord}} f^*(s_{\pi^{-1}1},\ldots,s_{\pi^{-1}m}) \to_{\text{mul}} f(r_1,\ldots,r_m) \to^*_{\mathcal{H}} f(t_1,\ldots,t_m)$, where $r_i = s_{\pi^{-1}i} = t_i$ if i is one of the j_x, $s_n \to_{\text{put}} r_i \to^*_{\mathcal{H}} t_i$ otherwise.

For the case $|B| > 1$, we observe that a comparison $X >_{\text{MUL}} Y$ can always be decomposed into a sequence $X = X_1 \rightsquigarrow X_2 \rightsquigarrow \ldots \rightsquigarrow X_n = Y$ where each $X_i \rightsquigarrow X_{i+1}$ is an atomic $>_{\text{MUL}}$ step as described above, and that $\to^*_{\mathcal{H}}$ is transitive.

(H4) We have $s = f(s_1,\ldots,s_m)$, $t = @(t_1 : \rho_1,\ldots,t_n : \rho_n)$ with $n \geq 2$, and $s \succ\!\succ \{t_1,\ldots,t_n\}$. By an analysis as in (H2), we can define for every $i \in \{1,\ldots,n\}$ a term l_i such that $l_i \to^*_{\mathcal{H}} t_i$. Note that $\rho_1 = \rho_2 \to \ldots \to \rho_n \to \sigma$. Therefore, $s \to_{\text{put}} f^*(s_1,\ldots,s_m) \to_{\text{appl}} @(l_1,\ldots,l_n) \to^*_{\mathcal{H}} @(t_1,\ldots,t_n)$.

(H5) We have $s = @(s_1,s_2)$, $t = @(t_1,t_2)$ and $\{\{s_1,s_2\}\} \succ_{MUL} \{\{t_1,t_2\}\}$.

Because of the typing constraints either $s_1 \succ t_1$ and $s_2 = t_2$ or $s_1 \succeq t_1$ and $s_2 \succ t_2$. In the first case, we have by the IH $s_1 \to_{\text{put}} s_1' \to^*_{\mathcal{H}} t_1$. Therefore, $s \to_{\text{put}} @(s_1',s_2) \to^*_{\mathcal{H}} t_1 s_2 = t$. In the second case, we have by the IH $s_2 \to_{\text{put}} s_2' \to^*_{\mathcal{H}} t_2$. Additionally, $s_1 \to^*_{\mathcal{H}} t_1$, although this may be in 0 steps. Therefore, $s \to_{\text{put}} @(s_1,s_2') \to^*_{\mathcal{H}} @(s_1,t_2) \to^*_{\mathcal{H}} @(t_1,t_2) = t$.

(H6) We have $s = \lambda x : \sigma.\, s_0$, $t = \lambda x : \sigma.\, t_0$, and $s_0 \succ t_0$. By the IH, $s_0 \to_{\text{put}} s_0' \to^*_{\mathcal{H}} t_0$. Hence $s = \lambda x : \sigma.\, s_0 \to_{\text{put}} \lambda x : \sigma.\, s_0' \to^*_{\mathcal{H}} \lambda x : \sigma.\, t_0 = t$. □

7 HORPO Does Not Contain HOIPO

In this section we present a rewrite rule $l \to r$ for which $l \succ^+ r$ in HORPO does not hold, but for which we can prove $l \to^+_{\mathcal{H}\beta} r$. This shows that HORPO does not contain HOIPO; the crux is that postponing the choice for a smaller term can sometimes be useful.

We consider the the following rewrite rule $l \to r$, using function symbols $G, H : o \to o \to o$ and $A, B : o$ and $f : (o \times o \to o) \to o$ (so f is the only function symbol that takes arguments):

$$f(B, \lambda z : o.\, @(G,z,z)) \to @(G, f(B, \lambda z : o.\, @(H,z,z)), f(A, \lambda z : o.\, @(G,z,z)))$$

In addition assume that there are rewrite rules that enforce $G \rhd H$ and $f \rhd B \rhd A$. We have $l \to^+_{\mathcal{H}\beta} r$:

$$\begin{aligned}
& l = f(B, \lambda z : o.@(G,z,z)) \\
\to_{\text{put}}\ & f^*(B, \lambda z : o.\, @(G,z,z)) \\
\to_{\text{appl}}\ & @(\lambda z : o.\, @(G,z,z), f^*(B, \lambda z : o.\, @(G,z,z))) \\
\to_{\beta}\ & @(G, f^*(B, \lambda z : o.\, @(G,z,z)), f^*(B, \lambda z : o.\, @(G,z,z))) \\
\to_{\text{lex}}\ & @(G, f(B, \lambda z : o.\, @(G^*,z,z)), f^*(B, \lambda z : o.\, @(G,z,z))) \\
\to_{\text{lex}}\ & @(G, f(B, \lambda z : o.\, @(G^*,z,z)), f(B^*, \lambda z : o.\, @(G,z,z))) \\
\to_{\text{copy}}\ & @(G, f(B, \lambda z : o.\, @(H,z,z)), f(B^*, \lambda z : o.\, @(G,z,z))) \\
\to_{\text{copy}}\ & @(G, f(B, \lambda z : o.\, @(H,z,z)), f(A, \lambda z : o.\, @(G,z,z))) = r
\end{aligned}$$

It is easy to see that $l \succ r$ does not hold. But do we have $l \succ^+ r$? We show that this is not the case. Suppose that $l \succ^+ t$ for some term t. We can prove by induction, first over the length of the $\succ$-sequence, second on the size of t, that t must have one of the following forms:

(a) $f(A, \lambda z : o.@(L, z, z))$ with $L \in \{G, H\}$
(b) $f(B, \lambda z : o.@(H, z, z))$
(c) $@(\lambda z : o.@(L, z, z), K)$ with $L \in \{G, H\}$ and $l \succ^+ K$
(d) $@(L, K_1, K_2)$ with $L \in \{G, H\}$ and there exists some K such that $l \succ^+ K$ and $K \succ^* K_1, K \succ^* K_2$.

The reason that there are so few possibilities is that A and H are minimal with respect to $\succ$, and B, G only compare to A, H.

Now, r has form (d). So there has to be some K of one of the forms above such that $K \succ^* K_1, K \succ^* K_2$. However, using induction we can see that whenever $s \succ t$ it can not hold that t contains f, G or B without s containing that symbol too. So if K has form (a) it can not reduce to a term with B in it, form (b) will not reduce to a term with G in it, and the last two forms will never reduce to a term with G in it.

So we see that following the same idea as in the iterative version does not work because there is no term t satisfying the following three conditions: $f(B, \lambda z : o.@(G, z, z)) \succ^* t$, and $t \succ^* f(B, \lambda z : o.@(H, z, z))$, and $t \succ^* f(A, \lambda z : o.@(G, z, z))$.

A similar phenomenon seems to occur when comparing HOIPO to the stronger definition of $\succ$ as given in [2]. The following system can be proven to be termination using HOIPO. However, it does not seem to have an easy termination proof using $\succ$ as defined in [2]. The system consists of the following rewrite rules: $\{g(x, y, z) \to f(f(x, z), z), f(x, z) \to B, B \to A, f(B, \lambda x : o.g(x, x, z)) \to g(f(A, \lambda x : o.g(x, x, z)), f(B, \lambda x : o.B), z)\}$ using the function symbols $f : (o \times (o \to o)) \to o$, $g : (o \times o \times (o \to o)) \to o$, $B : o$, and $A : o$.

Finally, note that the problem illustrated above is easily solved by adding a pairing operator to the system which is smaller than the other function symbols.

8 Concluding Remarks

We have defined an iterative version of HORPO as defined in [6]. Other, more advanced, definitions of HORPO are given in [7,2]. We have on purpose chosen for the definition from [6], because it is conceptually and technically very clear; it even has been proof-checked in Coq [11]. Moreover, because it is the most basic variant, it is the best starting point for an investigation to stronger orderings. In fact, the present definition of HOIPO already lead to some ideas about strengthening the ordering, which will be developed in more detail.

Having said that, clearly the termination method provided by HOIPO is, just as the one provided by HORPO, rather weak. For instance, it cannot be used to prove termination of developments in the untyped λ-calculus, which can be done in a higher-order iterative ordering in longer versions of [10], following [13]. Therefore, in further work we intend to consider extensions of HOIPO which may

or may not be defined as iterative definitions of more sophisticated variants of HORPO. We then need to compare those extensions with the more sophisticated versions of HORPO [7,2], as well as with the long version of [10].

One of the natural next steps is to extend HOIPO using the notion of computable closure, to define iterative versions of HORPO for other frameworks of higher-order rewriting (CRSs or HRSs) [14], or to replace the ordering on function symbols by interpretations on terms.

Acknowledgements. We are very grateful to the referees of an earlier version and to the referees of LPAR 2008 for their remarks.

References

1. Bergstra, J., Klop, J.W.: Algebra of communicating processes. Theoretical Computer Science 37(1), 171–199 (1985)
2. Blanqui, F., Jouannaud, J.-P., Rubio, A.: The computability path ordering: The end of a quest. In: Kaminski, M., Martini, S. (eds.) CSL 2008. LNCS, vol. 5213, pp. 1–14. Springer, Heidelberg (2008)
3. Buchholz, W.: Proof-theoretic analysis of termination proofs. Annals of Pure and Applied Logic 75(1–2), 57–65 (1995)
4. Dershowitz, N.: Orderings for term rewriting systems. Theoretical Computer Science 17(3), 279–301 (1982)
5. Jouannaud, J.-P., Okada, M.: A computation model for executable higher-order algebraic specification languages. In: Proceedings of LICS 1991, Amsterdam, The Netherlands, pp. 350–361 (July 1991)
6. Jouannaud, J.-P., Rubio, A.: The higher-order recursive path ordering. In: Proceedings of LICS 1999, Trento, Italy, pp. 402–411 (July 1999)
7. Jouannaud, J.-P., Rubio, A.: Higher-order orderings for normal rewriting. In: Pfenning, F. (ed.) RTA 2006. LNCS, vol. 4098, pp. 387–399. Springer, Heidelberg (2006)
8. Jouannaud, J.-P., Rubio, A.: Polymorphic higher-order recursive path orderings. Journal of the ACM 54(1), 1–48 (2007)
9. Kamin, S., Lévy, J.-J.: Two generalizations of the recursive path ordering. University of Illinois (1980)
10. Klop, J.W., van Oostrom, V., de Vrijer, R.: Iterative lexicographic path orders. In: Futatsugi, K., Jouannaud, J.-P., Meseguer, J. (eds.) Algebra, Meaning, and Computation. LNCS, vol. 4060, pp. 541–554. Springer, Heidelberg (2006)
11. Koprowski, A.: Coq formalization of the higher-order recursive path ordering. Technical Report CSR-06-21, Eindhoven University of Technology (August 2006)
12. Nipkow, T.: Higher-order critical pairs. In: Proceedings of LICS 1991, Amsterdam, The Netherlands, pp. 342–349 (July 1991)
13. van Oostrom, V.: Personal communication (2008)
14. van Raamsdonk, F.: On termination of higher-order rewriting. In: Middeldorp, A. (ed.) RTA 2001. LNCS, vol. 2051, pp. 261–275. Springer, Heidelberg (2001)
15. Tait, W.W.: Intensional interpretations of functionals of finite type I. Journal of Symbolic Logic 32(2), 198–212 (1967)
16. Terese: Term Rewriting Systems. Cambridge Tracts in Theoretical Computer Science, vol. 55. Cambridge University Press, Cambridge (2003)

Author Index