Alessandro Armando
Peter Baumgartner
Gilles Dowek (Eds.)

LNAI 5195

Automated Reasoning

4th International Joint Conference, IJCAR 2008
Sydney, Australia, August 2008
Proceedings

Springer

Lecture Notes in Artificial Intelligence 5195

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

Subseries of Lecture Notes in Computer Science

Alessandro Armando Peter Baumgartner
Gilles Dowek (Eds.)

Automated Reasoning

4th International Joint Conference, IJCAR 2008
Sydney, Australia, August 12-15, 2008
Proceedings

Springer

Volume Editors

Alessandro Armando
Università di Genova
DIST Via le Causa 13
16145 Genova, Italy
E-mail: armando@dist.unige.it

Peter Baumgartner
NICTA Canberra
Tower A 7 London Circuit
Canberra ACT 2601, Australia
E-mail: Peter.Baumgartner@nicta.com.au

Gilles Dowek
École polytechnique
Laboratoire d'Infomatique LIX
91128 Palaiseau Cedex, France
E-mail: Gilles.Dowek@polytechnique.edu

Library of Congress Control Number: 2008931576

CR Subject Classification (1998): I.2.3, I.2, F.4.1, F.3, F.4, D.2.4

LNCS Sublibrary: SL 7 – Artificial Intelligence

ISSN 0302-9743
ISBN-10 3-540-71069-8 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-71069-1 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12446285 06/3180 5 4 3 2 1 0

Preface

This volume contains the papers presented at IJCAR 2008, the 4th International Joint Conference on Automated Reasoning, held August 12–15, 2008, in Sydney (Australia). The IJCAR conference series is aimed at unifying the different research principles within automated reasoning. IJCAR 2008 was the fusion of several major international events:

- **CADE:** The International Conference on Automated Deduction
- **FroCoS:** The Symposium on Frontiers of Combining Systems
- **FTP:** The Workshop on First-Order Theorem Proving
- **TABLEAUX:** The Conference on Analytic Tableaux and Related Methods

Previous versions of IJCAR were held in Seattle (USA) in 2006, Cork (Ireland) in 2004, and Siena (Italy) in 2001.

These proceedings comprise 4 contributions by invited speakers, 26 research papers, and 13 system descriptions. The volume also includes a short overview of the CASC-J4 competition for automated theorem proving systems that was conducted during IJCAR 2008. The invited speakers were Hubert Comon-Lundh, Nachum Dershowitz, Aarti Gupta, and Carsten Lutz. Their talks covered a broad spectrum of automated reasoning themes, viz., verification of security protocols, proof theoretical frameworks for first-order logic, automated decision procedures and software verification, and description logics.

The contributed papers were selected from 80 research paper submissions and 17 system description submissions. Each submission was reviewed by at least three reviewers, and decisions were reached after two weeks of discussion through an electronic Program Committee meeting. The submissions, reviews, and discussion were coordinated using the EasyChair conference management system. The accepted papers spanned a wide spectrum of research in automated reasoning, including saturation, equational reasoning and unification, automata-based methods, description logics and related logics, satifiability modulo theory, decidable logics, reasoning about programs, and higher-order logics.

The Herbrand Award for distinguished contributions to automated reasoning was presented to Edmund M. Clarke in recognition of his role in the invention of model checking and his sustained leadership in the area for more than two decades. The selection committee for the Herbrand Award consisted of the previous award winners of the last ten years, the trustees of CADE Inc., and the IJCAR 2008 Program Committee. The Herbrand award ceremony and the acceptance speech by Professor Clarke were part of the conference program.

In addition to the Program Committee and the reviewers, many people contributed to the success of IJCAR 2008. Geoff Sutcliffe served as the Publicity Chair and organized the systems competition, CASC-J4. The IJCAR Steering Committee consisted of Alessandro Armando, Franz Baader (Chair), Peter Baumgartner, Alan Bundy, Gilles Dowek, Rajeev Goré, Bernhard Gramlich, John Harrison, and

Ullrich Hustadt. Special thanks go to Andrei Voronkov for his EasyChair system, which makes many tasks of a Program Chair much easier.

We would like to thanks all people involved in organizing IJCAR 2008, as well as the sponsors the Australian National University, Intel, Microsoft Research, and NICTA.

May 2008

Alessandro Armando
Peter Baumgartner
Gilles Dowek

Conference Organization

Program Chairs

Alessandro Armando
Peter Baumgartner
Gilles Dowek

Program Committee

Christoph Benzmueller
Nikolaj Bjorner
Patrick Blackburn
Maria Paola Bonacina
Alessandro Cimatti
Roy Dyckhoff
Silvio Ghilardi
Jürgen Giesl
Rajeev Gore
Bernhard Gramlich
Reiner Hähnle
John Harrison
Deepak Kapur
Viktor Kuncak
Christopher Lynch
Tobias Nipkow
Hans de Nivelle
Nicola Olivetti
Lawrence Paulson
Silvio Ranise
Christophe Ringeissen
Albert Rubio
Michael Rusinowitch
Ulrike Sattler
Carsten Schürmann
Natarajan Shankar
Viorica Sofronie-Stokkermans
Geoff Sutcliffe
Cesare Tinelli
Ashish Tiwari
Luca Viganò
Andrei Voronkov
Toby Walsh
Frank Wolter

Conference Chair

Peter Baumgartner

Workshop and Tutorial Chair

Michael Norrish

Publicity Chair

Geoff Sutcliffe

Local Organization

Jinbo Huang, Michael Norrish, Andrew Slater, Toby Walsh

External Reviewers

Andreas Abel
Anbulagan
Takahito Aoto
Clark Barrett
Joachim Baumeister
Malgorzata Biernacka
Lars Birkedal
Thomas Bolander
Bianca Boretti
Olivier Bournez
Marco Bozzano
Paul Brauner
James Bridge
Björn Bringert
Chad Brown
Kai Bruennler
Roberto Bruttomesso
Richard Bubel
Serge Burckel
Guillaume Burel
Serenella Cerrito
Amine Chaieb
Ernie Cohen
Sylvain Conchon
Dominik Dietrich
Yu Ding
Lucas Dixon
Bruno Dutertre
Mnacho Echenim
Stephan Falke
Christian Fermüller
Camillo Fiorentini
Melvin Fitting
Pascal Fontaine
Alexander Fuchs
Martin Giese
Isabelle Gnaedig
Guillem Godoy
Alberto Griggio
James Harland
Emmanuel Hebrard
Thomas Hillenbrand
Dieter Hutter
Swen Jacobs
Barbara Jobstmann
Vladimir Komendantsky
Boris Konev
Konstantin Korovin
Shuvendu Lahiri
Stephane Lengrand
Giacomo Lenzi
Alexei Lisitsa
Thomas Lukasiewicz
Carsten Lutz
Michael Maher
Mark Marron
William McCune
George Metcalfe
Aart Middeldorp
Pierluigi Minari
Boris Motik
Leonardo de Moura
Jean-Yves Moyen
Peter Mueller
Cesar Munoz
Peter Müller
Enrica Nicolini
Robert Nieuwenhuis
Greg O'Keefe
Albert Oliveras
Nicola Olivetti
Jan Otop
Sam Owre
Ruzica Piskac
David Plaisted
Gian Luca Pozzato
Florian Rabe
Vincent Risch
Xavier Rival
Enric Rodriguez-Carbonell
Robert Rothenberg
Philipp Ruemmer
Gernot Salzer
Felix Schernhammer
Renate Schmidt
Thomas Schneider
Camilla Schwind
Roberto Sebastiani
Rob Shearer
John Slaney
Aaron Stump
Christino Tamon
Alwen Tiu
Stefano Tonetta
Duc-Khanh Tran
Kumar Neeraj Verma
Laurent Vigneron
Marco Volpe
Dirk Walther
Florian Widmann
Claus-Peter Wirth
Burkhart Wolff
Eric Wurbel
Jian Zhang
Daniele Zucchelli

Table of Contents

Software Verification: Roles and Challenges for Automatic Decision Procedures

Aarti Gupta

NEC Laboratories America, Inc.
4 Independence Way, Suite 200
Princeton, NJ 08520, USA

Software model checking has become popular with the successful use of predicate abstraction and refinement techniques to find real bugs in low-level C programs. At the same time, verification approaches based on abstract interpretation and symbolic execution are also making headway in real practice. Much of the recent progress is fueled by advancements in automatic decision procedures and constraint solvers, which lie at the computational heart of these approaches. In this talk, I will describe our experience with several of these verification approaches, highlighting the roles played by automatic decision procedures and their interplay with the applications. Specifically, I will focus on SAT- and SMT-based model checking, combined with abstract domain analysis based on polyhedra representations. I will also describe some challenges from software verification that can motivate interesting research problems in this area.

Proving Bounds on Real-Valued Functions with Computations

Guillaume Melquiond

INRIA–Microsoft Research joint center,
Parc Orsay Université, F-91893 Orsay Cedex, France
`guillaume.melquiond@inria.fr`

Abstract. Interval-based methods are commonly used for computing numerical bounds on expressions and proving inequalities on real numbers. Yet they are hardly used in proof assistants, as the large amount of numerical computations they require keeps them out of reach from deductive proof processes. However, evaluating programs inside proofs is an efficient way for reducing the size of proof terms while performing numerous computations. This work shows how programs combining automatic differentiation with floating-point and interval arithmetic can be used as efficient yet certified solvers. They have been implemented in a library for the Coq proof system. This library provides tactics for proving inequalities on real-valued expressions.

1 Introduction

In traditional formalisms, proofs are usually composed of deductive steps. Each of these steps is the instantiation of a logical rule or a theorem. While this may be well-adapted for manipulating logic expressions, it can quickly lead to inefficiencies when explicit computations are needed. Let us consider the example of natural numbers constructed from 0 and a successor function S. For example, the number 3 is represented by $S(S(S(0)))$. If one needs to prove that 3×3 is equal to 9, one can apply Peano's axioms, e.g. $a \times S(b) = a \times b + a$ and $a + S(b) = S(a+b)$, until 3×3 has been completely transformed into 9. The first steps of the proof are: $3 \times 3 = 3 \times 2 + 3 = (3 \times 1 + 3) + 3 = \ldots$ This proof contains about 15 instantiations of various Peano's axioms. Due to the high number of deductive steps, this approach hardly scales to more complicated expressions, even if more efficient representations of integers were to be used, e.g. radix-2 numbers.

While numerical computations are made cumbersome by a deductive approach, they can nonetheless be used in formal proofs. Indeed, type-theoretic checkers usually come with a concept of programs which can be expressed in the same language than the proof terms. Moreover, the formalism of these checkers assumes that replacing an expression $f(x)$ of a functional application by the result of the corresponding evaluation does not modify the truth of a statement. As a consequence, one can write computable recursive functions for addition `add` and multiplication `mul` of natural numbers. For instance, in a ML-like language, `mul` can be defined as:

```
let rec mul x = function
  | 0 -> 0
  | S y -> add (mul x y) x
```

This function is extensionally equal to Peano's multiplication. More precisely, the following theorem can be proved by recursive reasoning on y:

$$\texttt{mul\_spec} : \forall x\ \forall y \quad x \times y = \texttt{mul}\ x\ y.$$

Therefore, in order to prove the statement $3 \times 3 = 9$, the first step of the proof is an application of `mul_spec` in order to rewrite 3×3 as `mul` 3 3. So one has now to prove that `mul` 3 3 is equal to 9. This is achieved by simply evaluating the function. So the proof contains only one deductive step: the use of `mul_spec`. With this computational approach, the number of deductive steps depends on the number of arithmetic operators only; It does not depend on the size of the integers. As a matter of fact, one can go even further so that the number of deductive steps is constant, irrespectively of the complexity of the expressions. This is the approach presented in this article.

In the Coq proof assistant[1], the ability to use programs inside proofs is provided by the convertibility rule: Two convertible well-formed types have the same inhabitants. In other words, if p is a proof of a proposition A, then p is also a proof of any proposition B such that the type B is convertible to the type A. Terms – types are terms – are convertible if they have the same normal form with respect to β-reduction (and a few other Coq-related reductions). In particular, since `mul` 3 3 evaluates to 9 by β-reduction, the types/propositions `mul` $3\ 3 = 9$ and $9 = 9$ are convertible, so they have exactly the same inhabitants/proofs.

More generally, an unproven proposition $P(x, y, \ldots)$ is transformed into a sufficient proposition $f_P(x, y, \ldots) = \mathit{true}$ — the difficulty is in designing an efficient f_P function and proving it evaluates to *true* only when P holds. The proof system then checks if this boolean equality is convertible to $\mathit{true} = \mathit{true}$. If it is, then it has a trivial proof from which a proof of P can be deduced. Going from P to f_P is a single deductive step, so most of the verification time will be spent in evaluating f_P. Fortunately, the convertibility rule happens to be implemented quite efficiently in Coq [1,2], so it becomes possible to automatically prove some propositions on real numbers by simply evaluating programs. An example of such a proposition is the following one, where x and y are universally-quantified real numbers:

$$\frac{3}{2} \le x \le 2 \Rightarrow 1 \le y \le \frac{33}{32} \Rightarrow \left| \sqrt{1 + \frac{x}{\sqrt{x+y}}} - \frac{144}{1000} \times x - \frac{118}{100} \right| \le \frac{71}{32768}$$

In order to prove this logical proposition with existing formal methods, one can first turn it into an equivalent system of several polynomial inequalities. Then a resolution procedure, e.g. based on cylindrical algebraic decomposition [3] or on

[1] http://coq.inria.fr/

the *Nullstellensatz* theorem [4], will help a proof checker to conclude automatically. When the proposition involves elementary functions (e.g. cos, arctan, exp) in addition to algebraic expressions, the problem becomes undecidable. Some inequalities can still be proved by first replacing elementary functions by polynomial approximations [5,6]. A resolution procedure for polynomial systems can then complete the formal proof.

A different approach is based on interval arithmetic and numerical computations. The process inductively encloses sub-expressions with numbers and propagates these bounds until the range of all the expressions is known [7]. Naive interval arithmetic, however, suffer from a loss of correlation between the multiple occurrences of a variable. In order to avoid this issue, the problems can be split into several smaller problems or higher-order enclosures can be used instead [8,9].

This paper presents an implementation of this approach for Coq. It will focus on the aspects of automatic proof and efficiency. Section 2 describes the few concepts needed for turning numerical computations and approximations into a formal tool. Section 3 describes the particularities of the datatypes and programs used for these computations. Section 4 finally brings those components together in order to provide fully-automatized "tactics" (the tools available to the user of Coq).

2 Mathematical Foundations

While both terms "computations" and "real numbers" have been used in the introduction, this work does not involve computational real numbers, e.g. sequences of converging rational numbers. As a matter of fact, it is based on the standard Coq theory of real numbers, which is a pure axiomatization with no computational content.

Section 2.1 presents the extension of real numbers defined for this work. This extension is needed in order to get a simpler definition of differentiation. Section 2.2 describes the interval arithmetic formalized in order to bound expressions. Section 2.3 then shows how differentiation and intervals are combined in order to tighten the computed bounds.

2.1 Extended Real Numbers

The standard theory of Coq describes real numbers as a complete Archimedean field. Since functions of type $\mathbb{R} \to \mathbb{R}$ are total, this formalism makes it a bit troublesome to deal with mathematical partial functions. For instance, $x \mapsto \frac{x}{x}$ is 1 for any real $x \neq 0$. For $x = 0$, the function is defined and its value is 0 in Coq. Indeed, $0 \cdot 0^{-1} = 0$, since 0^{-1} is the result of a total function, hence real. Similarly, one can prove, that the derivative of this function is always zero. So we have a function that seems both constant and discontinuous at $x = 0$.

Fortunately, this does not induce a contradiction, since it is impossible to prove that the derivative (the variation limit) of $x \mapsto \frac{x}{x}$ was really 0 at point 0. The downside is that, every time one wants to use the value of the derivative function, one has to prove that the original function is actually derivable. This requires to carry lots of proof terms around, which will prevent a purely computational approach.

So an element $\perp$ is added to the formalism in order to represent the "result" of a function outside its definition domain. In the Coq development, this additional element is called *NaN* (Not-a-Number) as it shares the same properties as the *NaN* value from the IEEE-754 standard on floating-point arithmetic [10]. In particular, *NaN* is an absorbing element for all the arithmetic operators on the extended set $\overline{\mathbb{R}} = \mathbb{R} \cup \{\perp\}$. That way, whenever an intermediate result is undefined, a $\perp$ value is propagated till the final result of the computation.

Functions of $\overline{\mathbb{R}} \to \overline{\mathbb{R}}$ can then be created by composing these new arithmetic operators. In order to benefit from the common theorems of real analysis, the functions are brought back into $\mathbb{R} \to \mathbb{R}$ by applying a projection operator:

$$\mathit{proj}_a : f \in (\overline{\mathbb{R}} \to \overline{\mathbb{R}}) \mapsto \left(x \in \mathbb{R} \mapsto \begin{cases} a & \text{if } f(x) = \perp \\ f(x) & \text{otherwise} \end{cases} \right) \in (\mathbb{R} \to \mathbb{R})$$

Then, an extended function f is defined as continuous at a point $x \neq \perp$ if $f(x) \neq \perp$ and if all the projections $\mathit{proj}_a(f)$ are continuous at point x. Similarly, f is defined as derivable at $x \neq \perp$ if $f(x) \neq \perp$ and if all the projections of f have the same derivative $d_f(x)$ at point x. A function f' is then a derivative of f if $f'(x) = d_f(x)$ whenever $f'(x) \neq \perp$.

From these definitions and standard analysis, it is easy to formally prove some rules for building derivatives. For example, if f' and g' are some derivatives of f and g, then the extended function $(f' \times g - g' \times f)/g^2$ is a derivative of f/g. This is true even if g evaluates to 0 at a point x, since the derivative would then evaluate to $\perp$ at this point.

As a consequence, if the derivative f' of an expression f does not evaluate to $\perp$ on the points of a connected set of $\mathbb{R}$, then f is defined, continuous, and derivable on all the points of this set, when evaluated on real numbers. The important point is that the extended derivative f' can be automatically built from an induction on the structure of the function f, without having to prove that f is derivable on the whole domain.

2.2 Interval Arithmetic

All the *intervals* (closed connected subsets) of $\mathbb{R}$ can be represented as pairs of extended numbers[2]:

$$\begin{aligned} \langle \perp , \perp \rangle &\mapsto \mathbb{R} \\ \langle \perp , u \rangle &\mapsto \{x \in \mathbb{R} \mid x \leq u\} \\ \langle l , \perp \rangle &\mapsto \{x \in \mathbb{R} \mid l \leq x\} \\ \langle l , u \rangle &\mapsto \{x \in \mathbb{R} \mid l \leq x \leq u\} \end{aligned}$$

[2] The pair $\langle \perp, \perp \rangle$ traditionally represents the empty set. This is the opposite here, as $\langle \perp, \perp \rangle$ means $\mathbb{R}$. This is a consequence of having no infinities in the formalization: $\perp$ on the left means $-\infty$ while $\perp$ on the right means $+\infty$. The empty set is represented by any pair $\langle l, u \rangle$ with $l > u$.

This set $\mathbb{I}$ of intervals is extended with a *NaI* (Not-an-Interval): $\overline{\mathbb{I}} = \mathbb{I} \cup \{\bot_I\}$. This new interval $\bot_I$ represents the set $\overline{\mathbb{R}}$. In particular, this is the only interval that contains $\bot$. As a consequence, if the value of an expression on $\overline{\mathbb{R}}$ is contained in an interval $\langle l, u\rangle$, then this value is actually a real number. This implies that the expression is well-formed on the real numbers and that it is bounded by l and u.

Interval Extensions and Operators. A function $F \in \overline{\mathbb{I}} \to \overline{\mathbb{I}}$ is defined as an interval extension of a function $f \in \overline{\mathbb{R}} \to \overline{\mathbb{R}}$, if

$$\forall X \in \mathbb{I}, \forall x \in \overline{\mathbb{R}}, \quad x \in X \Rightarrow f(x) \in F(X).$$

This definition can be adapted to non-unary functions too. An immediate property is: The result of $F(X)$ is $\bot_I$ if there exists some $x \in X$ such that $f(x) = \bot$. Another property is the compatibility of interval extension with function composition: If F and G are extensions of f and g respectively, then $F \circ G$ is an extension of $f \circ g$.

Again, an interval extension will be computed by an induction on the structure of an expression. This requires interval extensions of the arithmetic operators and elementary functions. For instance, addition and subtraction are defined as propagating $\bot_I$ and verifying the following rules:

$$\langle l_1, u_1\rangle + \langle l_2, u_2\rangle = \langle l_1 + l_2, u_1 + u_2\rangle$$
$$\langle l_1, u_1\rangle - \langle l_2, u_2\rangle = \langle l_1 - u_2, u_1 - l_2\rangle$$

Except for the particular case of $\bot$ meaning an infinite bound, this is traditional interval arithmetic [11,12]. So extending the other arithmetic operators does not cause much difficulty. For instance, if l_1 is negative and if both u_1 and l_2 are positive, the result of the division $\langle l_1, u_1\rangle / \langle l_2, u_2\rangle$ is $\langle l_1/l_2, u_1/l_2\rangle$.

Notice that this division algorithm depends on the ability to decide the signs of the bounds. More generally, one has to compare bounds when doing interval arithmetic. Section 3.1 solves it by restricting the bounds to a subset of $\overline{\mathbb{R}}$.

2.3 Bounds on Real-Valued Functions

Properties described in previous sections can now be mixed together for the purpose of bounding real-valued functions. Let us consider a function f of $\overline{\mathbb{R}} \to \overline{\mathbb{R}}$, for which we want to compute an interval enclosure Y such that $f(x) \in Y$ for any x in the interval $X \neq \bot_I$. Assuming we have an interval extension F of f, then the interval $F(X)$ is an acceptable answer.

Unfortunately, if X appears several times in the unfolded expression of $F(X)$, loss of correlation occurs: The wider the interval X is, the poorer the bounds obtained from $F(X)$ are. The usual example is $f(x) = x - x$. It always evaluates to 0, but its trivial extension $F(X) = X - X$ evaluates to $\langle 0, 0\rangle$ only when X is a singleton.

Monotone Functions. Let us suppose now that we also have an interval extension F' of a derivative f' of f. By definitions of interval extension and derivability, if $F'(X)$ is not $\perp_I$, then f is continuous and derivable at each point of X.

Moreover, if $F'(X)$ does not contain any negative value, then f is an increasing function on X. If X has real bounds l and u, then an enclosure of f on X is the convex hull $F(\langle l,l\rangle) \vee F(\langle u,u\rangle)$. As the interval $\langle l,l\rangle$ contains one single value when $l \neq \perp$, the interval $F(\langle l,l\rangle)$ should not be much bigger than the set $\{f(l)\}$ for any F that is a reasonable interval extension of f. As a consequence, $F(\langle l,l\rangle) \vee F(\langle u,u\rangle)$ should be a tight enclosure of f on X. The result is identical if $F'(X)$ does not contain any positive values.

First-Order Interval Evaluation. When $F'(X)$ contains both positive and negative values, there are no methods giving sharp enclosure of f. Yet $F'(X)$ can still be used in order to find a better enclosure than $F(x)$. Indeed, variations of f on X are bounded:

$$\forall a, b \in X\ \exists c \in X \quad f(b) = f(a) + (b - a) \cdot f'(c).$$

Once translated to intervals, this gives the following theorem:

$$\forall a, b \in X \quad f(b) \in F(\langle a,a\rangle) + (X - \langle a,a\rangle) \cdot F'(X).$$

As $F'(X)$ may contain even more occurrences of X than $F(X)$ did, the loss of correlation may be worse when computing an enclosure of f' than an enclosure of f. From a numerical point of view, however, we have more leeway. Indeed, the multiplication by $X - \langle a,a\rangle$, which is an interval containing only "small" values around zero, will mitigate the loss of correlation. This approach to proving bounds on expressions can be generalized to higher-order derivatives by using Taylor models [9].

3 Computational Datatypes

Proving propositions by computations requires adapted data types. This work relies on floating-point arithmetic (Section 3.1) for numerical computations and on straight-line programs (Section 3.2) for representing and evaluating expressions.

3.1 Floating-Point Arithmetic

Since interval arithmetic suffers from loss of correlation, the bounds are usually not sharp, so they do not need to be represented with exact real numbers. As a consequence, an interval extension does not need to return the "best" bounds. Simpler ones can be used instead. An interesting subset of $\overline{\mathbb{R}}$ is the set $\overline{\mathbb{F}} = \mathbb{F} \cup \{\perp\}$ of radix-2 floating-point numbers. Such a number is a rational number that can be written as $m \cdot 2^e$ with m and e integers.

Rounding. Let us consider the non-$\perp$ quotient $\frac{u}{v}$ of two floating-point numbers. This quotient is often impossible to represent as a floating-point number. If this value is meant to be the lower bound of an interval quotient, we can chose any floating-point number $m \cdot 2^e$ less than the ideal quotient. Among these numbers, we can restrict ourselves to numbers with a mantissa m represented with less than p bits (in other words, $|m| < 2^p$). There is an infinity of such numbers. But one of them is bigger than all the others. This is what the IEEE-754 standard calls the result of u/v rounded toward $-\infty$ at precision p.

Computing at fixed precision ensures that the computing time is linear in the number of arithmetic operations. Let us consider the computation of $\left(\frac{5}{7}\right)^{2^n}$ with n consecutive squaring. With rational arithmetic, the time complexity is then $O(n^3)$, as the size of the numbers double at each step. With floating-point arithmetic at fixed precision, the time complexity is just $O(n)$. The result is no longer exact, but interval arithmetic still works properly.

There have been at least two prior formalizations of floating-point arithmetic in Coq. The first one [13,14] defines rounded results with relations, so the value w would be expressed as satisfying the proposition:

$$w \le \frac{u}{v} \quad \land \quad \forall m, e \in \mathbb{Z}, \quad |m| < \beta^p \Rightarrow m \cdot \beta^e \le \frac{u}{v} \Rightarrow m \cdot \beta^e \le w$$

While useful and sufficient for proving theorems on floating-point algorithms, such a relation does not provide any computational content, so it cannot be used for performing numerical computations. The second formalization [15] has introduced effective floating-point operators, but only for addition and multiplication. The other basic operators are evaluated by an external oracle. The results can then be checked by the system with multiplications only. Elementary functions, however, cannot be reached with such an approach.

Implementation. In order to have effective floating-point operators that can be evaluated entirely inside Coq, this work needed a new formalization of floating-point arithmetic. The resulting library implements multi-radix[3] multi-precision operators for the four IEEE-754 rounding directions[4].

This library supports the basic arithmetic operators ($+$,$-$,$\times$, $\div$, $\sqrt{\cdot}$) and some elementary functions (arctan, cos, sin, tan, for now). Floating-point precision is a user-settable parameter of the automatic tactics. Its setting is a trade-off: A high precision can help in proving some propositions, but it also slows down computations.

In order to speed up floating-point computations by a $\times 10$ factor[5], the tactics do not use the standard integers of Coq, which are represented by lists of bits. They specialize the floating-point library so that it uses integers represented as

[3] Numbers are $m \cdot \beta^e$ for any integer $\beta \ge 2$. For interval arithmetic, the radix hardly matters. So the automatic tactics chose an efficient radix: $\beta = 2$.

[4] Only rounding toward $-\infty$ and $+\infty$ are needed when performing interval arithmetic.

[5] This speed-up is lower than one could expect. This is explained by the proofs not needing high-precision computations, so the mantissa integers are relatively small.

binary trees with leaves being radix-2^{31} digits [1]. The arithmetic on these leaves is then delegated by Coq to the computer processor [16].

3.2 Straight-Line Programs

Until now, we have only performed interval computations. We have yet to prove properties on expressions. A prerequisite is the ability to actually represent these expressions. Indeed, as we want Coq functions to be able to evaluate expressions in various ways, e.g. for bounds or for derivatives, they need a data structure containing an abstract syntax tree of the expressions. More precisely, an expression will be represented as a straight-line program. This is a directed acyclic graph with an explicit topological ordering on the nodes which contain arithmetic operators.

For example, the expression $\sqrt{x} - y \cdot \sqrt{x}$ is encoded as a sequence of three tuples representing the following straight-line program. Each tuple represents an arithmetic operation whose operands are the results of some of previous operations represented by a relative index – index 0 was computed at the previous step, index 1 two steps ago, and so on. The input values x and y are assumed to have already been computed by pseudo-operations with results in v_1 and v_0.

$$\begin{array}{ll} v_2 : (\text{sqrt}, 0) & \text{so } v_2 = \sqrt{v_{1-0}} = \sqrt{x} \\ v_3 : (\text{mul}, 2, 0) & \text{so } v_3 = v_{2-2} \cdot v_{2-0} = y \cdot \sqrt{x} \\ v_4 : (\text{sub}, 1, 0) & \text{so } v_4 = v_{3-1} - v_{3-0} = \sqrt{x} - y \cdot \sqrt{x} \end{array}$$

Notice that the square root occurs only once in the program. Representing expressions as straight-line programs makes it possible to factor common subexpressions. In particular, the computation of a given value (e.g. $\sqrt{x}$ here) will not be repeated several times during an evaluation.

The evaluation function is generic. It takes a list encoding the straight-line program, the type A of the inputs and outputs (e.g. $\overline{\mathbb{R}}$ or $\overline{\mathbb{I}}$), a record of functions implementing the operators (functions of type $A \to A$ and $A \to A \to A$), and a stack of inputs of type A. It then pushes on this stack the result of evaluating each operation stored in the list. It finally returns the stack containing the results of all the statements. Whenever an operation tries to access past the bottom of the evaluation stack, a default value of type A is used, e.g. 0 or $\perp$ or $\perp_I$.

When given various sets A and operators on A, the Coq function `eval` will produce, either an expression on real numbers corresponding to the straight-line program, or an interval enclosing the values of the expression, or an expression of the derivative of the expression, or bounds on this derivatives, etc. For instance, the derivative is bounded by evaluating the straight-line program on the set A of interval pairs – the first interval encloses the value of an expression, while the second one encloses the value of its derivative. The operators on A create these intervals with formulas related to automatic differentiation:

$$\begin{array}{rcl} \text{plus} & \equiv & (X, X') \in A \mapsto (Y, Y') \in A \mapsto (X + Y, X' + Y') \\ \text{mul} & \equiv & (X, X') \in A \mapsto (Y, Y') \in A \mapsto (X \times Y, X' \times Y + X \times Y') \\ \tan & \equiv & (X, X') \in A \mapsto (\tan X, X' \times (1 + \tan^2 X)) \\ & \ldots & \end{array}$$

4 Automatic Proofs

Now that we have the basic blocks, we can use the convertibility rule to build automatic tactics. First, convertibility helps transforming logical propositions into data structures on which programs can actually compute. Second, it gives a meaning to the subsequent numerical computations. The user does not have to worry about it though, since the tactics will take care of all the details.

4.1 Converting Terms

Let us assume that the user wants to prove $\sqrt{x} - y \cdot \sqrt{x} \leq 9$ knowing some bounds on x and y. As explained in Section 2.2, interval arithmetic will be used to bound the expression $\sqrt{x} - y \cdot \sqrt{x}$. The upper bound can then be compared to 9 in order to check that the expression was indeed smaller than this value. In order to perform this evaluation, the functions need the straight-line program representing the expression.

Unfortunately, this straight-line program cannot be built within Coq's term language, as the syntax of the expressions on real numbers is not available at this level. So the list representing the program has to be provided by an oracle.

Three approaches are available. First, the user could perform the transformation by hand. This may be fine for small terms, but it quickly becomes cumbersome. Second, one could implement the transformation directly into the Ocaml code of Coq, hence creating a new version of the proof assistant. Several existing reflexive tactics actually depend on Ocaml helpers embedded inside Coq, so this is not an unusual approach. Third, one could use the tactic language embedded in Coq [17], so that the transformation runs on an unmodified Coq interpreter. This third way is the one chosen for this work.

A Coq tactic will therefore parse the expression and create the program described in Section 3.2. While the expression has type $\mathbb{R}$, this program is a list. But when evaluated with the axiomatized operations on real numbers, the result should be the original expression, if the tactic did not make any mistake. The tactic also transforms the real number 9 into the floating-point number $+9 \cdot 2^0$. So the tactic tells Coq that proving the following equality is equivalent to proving the original inequality.

$$(\texttt{eval}\ \mathbb{R}\ \texttt{[Sqrt 0, Mul 2 0, Sub 1 0] [y, x]}) \leq +9 \cdot 2^0$$

Coq does not trust the tactic, so it will check that this transformation is valid. Here, both inequalities are convertible, so they have the same proofs. Therefore, the proof system just has to evaluate the members of the new inequality, in order to verify that the transformation is valid. This transformation process is called reflexion [18]: An oracle produces a higher-level representation of the user proposition, and the proof system has only to check that the evaluation of this better representation is convertible to the old one. This transformation does not involve any deductive steps; There is no rewriting steps with this approach, contrarily to the 3×3 example of the introduction.

4.2 Proving Propositions

At this point, the proposition still speaks about real numbers, since convertibility cannot modify what the proposition is about. So we need the following theorem in order to get a proposition about extended real numbers, hence suitable for interval arithmetic with automatic differentiation.

$$\begin{array}{l}\forall prog\ \forall inputs\ \forall l, u \\ (\texttt{eval}\ \overline{\mathbb{R}}\ \ prog\ \ inputs) \in \langle l, u\rangle \Rightarrow \\ (\texttt{eval}\ \mathbb{R}\ \ prog\ \ inputs) \in \langle l, u\rangle\end{array}$$

Since the interval operators are interval extensions of the arithmetic operators on $\overline{\mathbb{R}}$, and since interval extension is compatible with function composition, we also have the following theorem.

$$\begin{array}{l}\forall prog\ \forall inputs\ \forall ranges \\ (\forall j \quad inputs_j \in ranges_j) \Rightarrow \\ (\texttt{eval}\ \overline{\mathbb{R}}\ \ prog\ \ inputs) \in (\texttt{eval}\ \overline{\mathbb{I}}\ \ prog\ \ ranges)\end{array}$$

Let us assume there are hypotheses in the context that state $\texttt{x} \in \texttt{X}$ and $\texttt{y} \in \texttt{Y}$. By applying the two theorems above and the transitivity of interval inclusion, the tactic is left with the following proposition to prove:

$$(\texttt{eval}\ \overline{\mathbb{I}}\ \texttt{[Sqrt 0, Mul...] [Y, X]}) \subseteq \langle \bot, +9 \cdot 2^0\rangle$$

While the interval evaluation could be performed in this proposition, the interval inclusion cannot be verified automatically yet. In order to force the proof system to compare the bounds, a last theorem is applied, so that the inclusion is transformed into a boolean equality:

$$\texttt{subset}\ (\texttt{eval}\ \overline{\mathbb{I}}\ \texttt{[Sqrt 0, Mul...] [Y, X]})\ \langle \bot, +9 \cdot 2^0\rangle = true$$

The tactic then tells to Coq that this proposition is convertible to $true = true$. As comparisons between interval bounds are decidable, the `subset` function performs an effective computation, and so does `eval` on floating-point intervals. As a consequence, Coq is able to check the convertibility of these propositions by evaluating the left hand side of the equality. If the result of this evaluation is *true*, then the propositions have the same proofs. This conversion is numerically intensive and can take a long time, since it performs all the interval and floating-point computations. At this point, the tactic just has to remind Coq that equality is reflexive, so $true = true$ holds.

To summarize, this proof relies almost entirely on convertibility, except for a few deductive steps[6], which are instantiations of the following theorems:

1. If the result of a formula on extended reals is contained in a non-$\bot_I$ interval, then the formula on real numbers is well-formed and has the same bounds.

[6] In addition, some optional steps may be performed to simplify the problem. For instance, $|expr| \leq u$ is first transformed into $-u \leq expr \leq u$.

2. The interval evaluation of a given straight-line program is an interval extension of the evaluation of the same program on extended reals.
3. If `subset` $A\ B = \mathit{true}$, then any value contained in A is also contained in B.
4. Boolean equality is reflexive.

4.3 Bisection and Refined Evaluation

Actually, because of a loss of correlation, the left hand side evaluates to *false* on the example given in the introduction, so Coq complains that the proposition are not convertible. This is expected [8], and two methods experimented with the PVS proof assistant[7] can be reused here. The first method is the bisection: If the interval evaluation fails to return an interval small enough, split the input interval in two parts and perform the interval evaluation again on each of these parts. As the parts are smaller than the whole interval, the loss of correlation should be reduced, so the interval evaluation produces a sharper result [11].

In the PVS work, interval splitting is performed by applying a theorem for each sub-interval. Here we keep relying on programs in order to benefit from convertibility and reduce the number of deductive steps. The method relies on a `bisect` function recursively defined as:

```
bisect $n\ F\ \langle l,u\rangle\ \mathit{target}$ =
  if $n = 0$ then $\mathit{false}$
  else if (subset $F(\langle l,u\rangle)\ \mathit{target}$) then $\mathit{true}$
  else let $m$ be the midpoint of $\langle l,u\rangle$ in
    (bisect $(n-1)\ F\ \langle l,m\rangle\ \mathit{target}$)   &&
    (bisect $(n-1)\ F\ \langle m,u\rangle\ \mathit{target}$)
```

This function is meant to replace the `subset` call in the previous proof. Its associated theorem is a bit more complicated though, but its hypotheses are as easy to satisfy: If F is an interval extension of a function f and if `bisect` evaluates to *true*, then $f(x) \in \mathit{target}$ holds for any $x \in \langle l,u\rangle$. Once again, the complete proof contains only four deductive steps. Everything else is obtained through convertibility, with the evaluation of `bisect` being the numerically-intensive part of the proof.

As long as n is big enough and the floating-point computations are accurate enough, this method can solve most of the provable propositions of the form $\forall x \in \langle l,u\rangle,\ f(x) \in Y$ with f a straight-line program. The method is guaranteed[8] to succeed when l and u are finite and the sharpest enclosure of f on $\langle l,u\rangle$ is a subset of the interior of Y. The method can also be extended to multi-variate problems, by splitting interval boxes along each dimension iteratively.

The bigger the number of sub-intervals, the longer the proof will take. In order to keep this number small, the tactic calls `bisect` with an interval function F that performs first-order derivation, as described in Section 2.3.

[7] `http://pvs.csl.sri.com/`

[8] If there is $x \in \langle l,u\rangle$ such that $f(x)$ evaluates to $\bot$, Y has to be $\bot_I$. Otherwise, f is continuous on the compact set $\langle l,u\rangle$, hence it is also uniform continuous. This uniformity ensures that some suitable precision and some bisection depth exist.

5 Examples

The user-visible tactic is called `interval`. It can be used for proving inequalities involving real numbers. For instance, the following script is the Coq version of a PVS example proved by interval arithmetic [7]:

```
Goal
  let v := 250 * (514 / 1000) in
  3 * pi / 180 <= g / v * tan (35 * pi / 180).
Proof.
  apply Rminus_le. (* transform into a - b <= 0      *)
  interval.        (* prove by interval computations *)
Qed.
```

The strength of this work lies, however, in its ability to prove theorems on function approximations. Most mathematical functions are not available to developers, so they are usually replaced with approximations, e.g. truncated series. In order to certify that the programs are still valid after these transformations, one has to give and prove a bound on the error between the actual implementation and the ideal function. The absolute error is the difference between two close values (if the implementation is any good), which makes it hard to prove a tight bound on it — this is the $X - X$ issue of Section 2.3. The two examples below exercise the tactic on such ill-conditioned problems.

5.1 Remez' Polynomial of the Square Root

Taylor models have been experimented in Coq [9] in order to formally prove some inequalities of Hales' proof[9] of Kepler's conjecture. Part of the difficulty with Taylor models lies in handling elementary functions. Indeed, one has to use polynomial approximations for this purpose. Usually, as the name implies, these polynomials are Taylor expansions, since their expansion remainder can be bounded by symbolic methods. Yet Taylor expansions are poor approximations, so high-degree polynomials are needed, which needlessly slow down the proof.

There are much better polynomial approximations, e.g. the ones obtained from Remez' algorithm. Unfortunately, the approximation error is no longer available to symbolic methods. One has to bound it numerically. The following proposition states the error bound between the square root function and its Remez approximation of degree 5 with rational coefficients of width 20+20 bits, on the interval $0.5 \leq x \leq 2$:

$$\left| \left(\left(\frac{122}{7397} \times x - \frac{1733}{13547} \right) \times x + \cdots + \frac{227}{925} \right) - \sqrt{x} \right| \leq \frac{5}{65536}$$

Since Remez' algorithm returns the best polynomial approximation with real coefficients, checking the error bound is a numerically difficult problem. Yet it only takes a few seconds for the `interval` tactic to automatically prove it in Coq on a desktop computer. In comparison, the CAD algorithm (with fast integers too) needs more than ten minutes in Coq. For Hales' proof, one also needs the arctan function, which is in the scope of this tactic.

[9] `http://code.google.com/p/flyspeck/`

5.2 Relative Error for an Elementary Function

The following example is taken from another PVS proof [8]. In order to certify a numerical code, the objective was to prove a bound on the relative error between the following function r_p and the degree-10 polynomial $\hat{r}_p$ that approximated it.

$$r_p(\phi) = \frac{a}{\sqrt{1 + (1-f)^2 \cdot \tan^2 \phi}}$$

The relative error is defined as the quotient $\epsilon(\phi) = (r_p(\phi) - \hat{r}_p(\phi))/r_p(\phi)$ on the interval $0 \le \phi \le \frac{715}{512}$. Due to the loss of correlation, PVS interval strategies are unable to automatically prove the bound $23 \cdot 2^{-24}$ on this error. The problem could be split into smaller intervals, so that interval computations are able to prove the bound on each sub-interval. But the loss of correlation is extreme, so more than 10^6 sub-intervals are needed, which make it impossible to complete the whole proof in a reasonable amount of time. So first-order interval evaluation was performed (see Section 2.3) in order to bring the number of sub-intervals down to 10^5.

Unfortunately, several user actions are required with the PVS approach. First, the user has to prove beforehand that the relative error is well-formed (no division by zero, no square root of negative number, and so on). Second, the user has to prove the formulas involving the derivative. Third, an external oracle analyzes the proposition in order to choose good sub-intervals. It also searches the order at which the power series of tan have to be evaluated in order to provide results that are accurate enough. With these data, the oracle then generates 10^5 PVS scripts corresponding to all the sub-intervals, and one master script that states the theorem on the error bound. Fourth, the user dispatches all the generated scripts on the 48 cores of a parallel computer. A few hours later, proof verification is complete. Finally, PVS checks the master script: The bound $|\epsilon(\phi)| \le 24 \cdot 2^{-23}$ is formally proved.

Thanks to the work described in this article, the situation is now a lot more satisfying in Coq. The following script proves the same theorem (`arp` is the approximation polynomial) in a few minutes on the single core of a desktop computer.

```
Goal
  forall phi, 0 <= phi <= max ->
  Rabs ((rp phi - arp phi) / rp phi) <= 23/16777216.
Proof.
  unfold rp, arp, umf2, a, f, max. intros.
  interval with (i_bisect_diff phi, i_nocheck).  (* Time:  4s *)
Qed.                                             (* Time: 96s *)
```

The user has to tell the tactic on which variable to perform a bisection followed by a first-order evaluation: `i_bisect_diff phi`. The user could also tell at which precision the floating-point computations are performed and how deep the bisection should explore. But the default parameters, `i_prec 30` and `i_depth 15`, are sufficient for this proof.

All the details of the proof are then handled by the tactic. It first parses the proposition and creates the corresponding straight-line program. It then performs the four deductive steps and Coq is left with a boolean equality. The evaluation of this Coq term by the system will first cause an interval function that encloses the derivative of the straight-line program to be built. It will then launch the execution of an interval bisection until the expression is bounded on all the sub-intervals.

When the proof is achieved (at `Qed` time), Coq checks that the lambda-term corresponding to the whole proof is correctly typed. In particular, it means that the numerically-intensive convertibility is checked a second time. The `i_nocheck` parameter avoids this redundant computation: The convertibility check is no longer done at `interval` time, but only at `Qed` time. So the tactic needs 4 seconds for parsing the expressions and preparing the computations, and then `Qed` needs 96 seconds to actually perform them.

Because there are no oracles, the Coq proof performs at least twice as many numerical computations[10] and with a higher precision than needed. Yet, the proof verification is tremendously faster in Coq than in PVS, although the approach is similar. This improvement is explained by the underlying arithmetic. In PVS, the interval bounds are rational numbers, so the intermediate computations quickly involve integers with thousands of digits. In Coq, thanks to the floating-point rounded arithmetic, the integers are at most 62-bit long. So the computation time does not explode with the size of the expressions.

6 Conclusion

Interval-based methods have been used for the last thirty years whenever a numerical problem (bounding an expression, finding all the zeros of a function, solving a system of differential equations, and so on) needed to be solved in an efficient and reliable way. But due to their numerically-intensive nature, they have been seldom used within formal proofs. With the advent of fast program evaluation in proof checkers, the situation is evolving [15,9].

This work improves on the existing formal approaches by relying on an efficient underlying arithmetic. Indeed, the computations are performed thanks to an effective formalization of floating-point arithmetic, instead of relying on an arithmetic on rational numbers. This brings the formal proofs closer to the non-formal approaches that are based on numerical computations with floating-point numbers, e.g. Hales' original proof. Another improvement lies in a careful extension of the real analysis: All the internal notions are designed so that theorems can be entirely handled with computations. The user does not have to deal with fastidious details anymore, e.g. proofs of derivability.

As this work deals with automatic proofs of numeric bounds on real-valued expressions suffering from correlations, it seems closely related to the Gappa system [15]. There are two important differences though. First, Gappa is an external oracle that produces a deductive proof that has been optimized, while

[10] The bisection process can be seen as a binary tree. An inner node is an evaluation failure, which leads to an evaluation on two sub-intervals, its children. Leaves are successful interval evaluations, and they are the only nodes needed for the proof.

this work is embedded and produces a straightforward and computational proof. Second, Gappa is specially designed for non-continuous expressions with basic arithmetic operators and a strong underlying structure, while this work focuses on (infinitely-) derivable expressions with elementary functions.

While designed for different kinds of expressions, these two approaches are complementary when performing formal certification of numerical applications. For instance, Section 5.2 was replacing an elementary function r_p with a polynomial $\hat{r}_p$. This polynomial is meant to be evaluated by a processor, but this evaluation $\tilde{r}_p$ will suffer from rounding errors. The implementation will be useful, only if the computed value $\tilde{r}_p(\phi)$ is close enough to the mathematical value $r_p(\phi)$. This certification is usually performed by separately bounding the distances between $\tilde{r}_p$ and $\hat{r}_p$, and between $\hat{r}_p$ and r_p. The first bound can be proved by Gappa but not by this work, since the expression is non-continuous. The second bound, however, can be proved by this work but not by Gappa, since the expression contains trigonometric terms and it has to be automatically differentiated for efficiency.

This work computes first-order derivatives of the expressions. This is usually sufficient for handling the correlations that appear when certifying numerical applications in most embedded systems, e.g. with a relative error of magnitude 2^{-25}. It will, however, fail to prove longer approximations which have a much higher accuracy. Therefore, tools that access higher-order derivatives through Taylor series [6,9] should perform much better in these latter cases. In case of high-dimension input domains, the approach presented in this article will also perform worse than multi-variate Bernstein polynomials [9].

Therefore, this work should not be seen as a panacea for proving inequalities on real-valued expressions without any user interaction. It is more of a proof-of-concept that shows how some numerical methods (floating-point arithmetic and interval arithmetic) can be combined with the convertibility rule in order to formally prove theorems. The generated proof terms contain almost no deductive steps and the tactic is nothing more than a parser, yet this approach is able to automatically prove arbitrarily-complicated inequalities.

The Coq development presented in this paper is available at

`http://www.msr-inria.inria.fr/~gmelquio/soft/coq-interval/`

References

1. Grégoire, B., Théry, L.: A purely functional library for modular arithmetic and its application to certifying large prime numbers. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 423–437. Springer, Heidelberg (2006)
2. Grégoire, B., Théry, L., Werner, B.: A computational approach to Pocklington certificates in type theory. In: Hagiya, M., Wadler, P. (eds.) FLOPS 2006. LNCS, vol. 3945, pp. 97–113. Springer, Heidelberg (2006)
3. Mahboubi, A.: Implementing the cylindrical algebraic decomposition within the Coq system. Mathematical Structure in Computer Sciences 17(1) (2007)
4. Harrison, J.: Verifying nonlinear real formulas via sums of squares. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 102–118. Springer, Heidelberg (2007)

5. Harrison, J.: Floating point verification in HOL light: The exponential function. In: Algebraic Methodology and Software Technology, 246–260 (1997)
6. Akbarpour, B., Paulson, L.C.: Towards automatic proofs of inequalities involving elementary functions. In: Cook, B., Sebastiani, R. (eds.) PDPAR: Pragmatics of Decision Procedures in Automated Reasoning, pp. 27–37 (2006)
7. Muñoz, C., Lester, D.: Real number calculations and theorem proving. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 195–210. Springer, Heidelberg (2005)
8. Daumas, M., Melquiond, G., Muñoz, C.: Guaranteed proofs using interval arithmetic. In: Montuschi, P., Schwarz, E. (eds.) Proceedings of the 17th IEEE Symposium on Computer Arithmetic, Cape Cod, MA, USA, pp. 188–195 (2005)
9. Zumkeller, R.: Formal global optimisation with Taylor models. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 408–422. Springer, Heidelberg (2006)
10. Stevenson, D., et al.: An American national standard: IEEE standard for binary floating point arithmetic. ACM SIGPLAN Notices 22(2), 9–25 (1987)
11. Moore, R.E.: Methods and Applications of Interval Analysis. SIAM, Philadelphia (1979)
12. Jaulin, L., Kieffer, M., Didrit, O., Walter, E.: Applied Interval Analysis, with Examples in Parameter and State Estimation, Robust Control and Robotics. Springer, Heidelberg (2001)
13. Daumas, M., Rideau, L., Théry, L.: A generic library of floating-point numbers and its application to exact computing. In: Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics, Edinburgh, Scotland, pp. 169–184 (2001)
14. Boldo, S.: Preuves formelles en arithmétiques à virgule flottante. PhD thesis, École Normale Supérieure de Lyon (2004)
15. Melquiond, G.: De l'arithmétique d'intervalles à la certification de programmes. PhD thesis, École Normale Supérieure de Lyon, Lyon, France (2006)
16. Spiwack, A.: Ajouter des entiers machine à Coq. Technical report (2006)
17. Delahaye, D.: A tactic language for the system Coq. In: Parigot, M., Voronkov, A. (eds.) LPAR 2000. LNCS (LNAI), vol. 1955, pp. 85–95. Springer, Heidelberg (2000)
18. Boutin, S.: Using reflection to build efficient and certified decision procedures. In: Theoretical Aspects of Computer Software, pp. 515–529 (1997)

Linear Quantifier Elimination

Tobias Nipkow

Institut für Informatik, Technische Universität München

Abstract. This paper presents verified quantifier elimination procedures for dense linear orders (DLO), for real and for integer linear arithmetic. The DLO procedures are new. All procedures are defined and verified in the theorem prover Isabelle/HOL, are executable and can be applied to HOL formulae themselves (by reflection).

1 Introduction

This paper is about the concise implementation of *quantifier elimination* (QE) procedures (QEPs) for linear arithmetics. QE is a venerable logical technique which yields decision procedures if ground atoms are decidable. The focus of our work is the compact implementation of QEPs (for linear arithmetics) inside a theorem prover. All our QEPs have been defined and verified in Isabelle/HOL [16]. We do not discuss these formal proofs here. They are detailed, mostly structured and available online at `afp.sf.net`, together with the QEPs themselves. Because the informal proofs of these QEPs can be found in the literature, they need not be discussed either. The exception are our two new QEPs for which informal correctness proofs are given.

The main contributions of this paper are:

– Two new QEPs for dense linear orders (DLO) inspired by QEPs for linear real arithmetic.
– Presentation of 5 verified implementations of QEPs: two for DLO, two for linear real arithmetic and one for Presburger arithmetic (Cooper). We show everything but the most trivial details, providing reference implementations and convincing the reader that nothing has been swept under the carpet.
– Extremely compact formalizations due to the almost excessive use of lists and list comprehensions.
– A common reusable QE framework using Isabelle's structuring facility of locales, thus factoring out the common parts of the different QEPs.

Why this obsession with executable and verified QEPs? The context of this research is the question of how to implement trustworthy and efficient decision procedures in foundational theorem provers, i.e. without having to trust an external oracle. *Reflection*, originally proposed by Boyer and Moore [2] and used to great effect in systems like Coq (e.g. [7]) and Isabelle (e.g. [4]) has become a standard approach. Suffice it to say that we follow this approach, too, and that all the algorithms in this paper can be used directly on formulae in Isabelle — details can be found elsewhere (e.g. [15]).

This paper is a contribution to the growing body of verified theorem proving algorithms. In spirit it is close to Harrison's forthcoming book [9] which presents all algorithms in OCaml. Only that our code is verified.

It should be emphasized that the presentation is streamlined for succinctness. In particular, we always restrict attention to two of the four relations $=$, $<$, $\leq$, $\neq$. For example, in DLOs it suffices to consider $=$ and $<$ because $x \leq y$ is equivalent with $x < y \vee x = y$ and $x \neq y$ is equivalent with $x < y \vee y < x$. For QEPs based on DNF this is a disaster because it leads to further case splits. The algorithms in this paper avoid DNF. Nevertheless, an efficient implementation would always work with all four relations. The corresponding generalization of our code is straightforward.

The paper is structured as follows. In §3 we describe a HOL model of logical formulae parameterized by a language of atoms and present a generic QEP parameterized by a QEP for a single quantifier. The remaining sections present a succession of 5 single-quantifier QEPs for different linear theories.

2 Basic Notation

HOL conforms largely to everyday mathematical notation. This section introduces further non-standard notation and in particular a few basic data types with their primitive operations.

The types of truth values, natural numbers, integers and reals are called *bool*, *nat*, *int* and *real*. The space of total functions is denoted by $\Rightarrow$. Type variables are denoted by α, β, etc. The notation $t{::}\tau$ means that term t has type τ.

Sets over type α, type α *set*, follow the usual mathematical convention.

Lists over type α, type α *list*, come with the empty list $[]$, the infix constructor $\cdot$, the infix @ that appends two lists, and the conversion function *set* from lists to sets. Variable names ending in s usually stand for lists. In addition to the standard functions *map* and *filter*, Isabelle/HOL also supports Haskell-style list comprehension notation, with minor differences: instead of `[e | x <- xs, ...]` we write $[e.\ x \leftarrow xs, \ldots]$, and $[x{\leftarrow}xs.\ \ldots]$ is short for $[x.\ x{\leftarrow}xs, \ldots]$.

Finally note that $=$ on type *bool* means "iff".

During informal explanations we often switch to everyday mathematical notation where (a, b) can be a pair or an open interval.

3 Logic

Formulae are defined as a recursive datatype with a parameter type α of atoms:

$$\textbf{datatype}\ \alpha\ \mathit{fm} = \top \mid \bot \mid A\ \alpha$$
$$\mid\ (\alpha\ \mathit{fm})\ \pmb{\wedge}\ (\alpha\ \mathit{fm}) \mid (\alpha\ \mathit{fm})\ \pmb{\vee}\ (\alpha\ \mathit{fm}) \mid \pmb{\neg}\ (\alpha\ \mathit{fm}) \mid \pmb{\exists}\ (\alpha\ \mathit{fm})$$

The **boldface** symbols $\pmb{\wedge}$, $\pmb{\vee}$, $\pmb{\neg}$ and $\pmb{\exists}$ are ordinary constructors chosen to resemble the logical operators they represent. Constructor A encloses atoms. The type of atoms is left open by making it a parameter α. Variables are represented by de

Bruijn indices: quantifiers do not explicitly mention the name of the variable being bound because that is implicit. For example, $\exists\ (\exists\ \ldots\ \mathit{0}\ \ldots\ \mathit{1}\ \ldots)$ represents a formula $\exists x_1 . \exists x_0 .\ \ldots x_0 \ldots x_1 \ldots$. Note that the only place where variables can appear is inside atoms. The only distinction between free and bound variables is that the index of a free variable is larger than the number of enclosing binders.

3.1 Auxiliary Functions

The constructors $\vee$, $\wedge$ and $\neg$ have optimized ("short-circuit") versions *or*, *and* and *neg*: $\mathit{or}\ \top\ \varphi = \top$, $\mathit{or}\ \varphi\ \top = \top$, $\mathit{or}\ \bot\ \varphi = \varphi$, $\mathit{or}\ \varphi\ \bot = \varphi$ and $\mathit{or}\ \varphi_1\ \varphi_2 = (\varphi_1 \vee \varphi_2)$ otherwise; $\mathit{and}\ \top\ \varphi = \varphi$, $\mathit{and}\ \varphi\ \top = \varphi$, $\mathit{and}\ \bot\ \varphi = \bot$, $\mathit{and}\ \varphi\ \bot = \bot$ and $\mathit{and}\ \varphi_1\ \varphi_2 = (\varphi_1 \wedge \varphi_2)$ otherwise; $\mathit{neg}\ \top = \bot$, $\mathit{neg}\ \bot = \top$ and $\mathit{neg}\ \varphi = \neg\ \varphi$ otherwise.

Disjunction of a lists of formulae is easily defined:

$\mathit{list\text{-}disj}\ [\varphi_1,\ldots,\varphi_n] = \mathit{or}\ \varphi_1\ (\mathit{or}\ \ldots\ \varphi_n)$

Most of our work will be concerned with quantifier-free formulae where all negations have not just been pushed right in front of atoms but actually into them. This is easy for linear orders because $\neg(x < y)$ is equivalent with $y \leq x$. This conversion will be described later on because it depends on the type of atoms. The (trivial to define) predicates

$$\mathit{qfree},\ \mathit{nqfree} :: \alpha\ \mathit{fm} \Rightarrow \mathit{bool}$$

check whether their argument is free of quantifiers (*qfree*), and free of negations and quantifiers (*nqfree*).

There are also two mapping functionals

$$\begin{aligned} \mathit{map}_{\mathit{fm}}\ &:: (\alpha \Rightarrow \beta) \Rightarrow \alpha\ \mathit{fm} \Rightarrow \beta\ \mathit{fm} \\ \mathit{amap}_{\mathit{fm}} &:: (\alpha \Rightarrow \beta\ \mathit{fm}) \Rightarrow \alpha\ \mathit{fm} \Rightarrow \beta\ \mathit{fm} \end{aligned}$$

where $\mathit{map}_{\mathit{fm}}\ f$ is the canonical one that simply replaces $A\ a$ by $A\ (f\ a)$, whereas $\mathit{amap}_{\mathit{fm}}$ may also simplify the formula via *and*, *or* and *neg*:

$\mathit{amap}_{\mathit{fm}}\ h\ \top = \top \qquad \mathit{amap}_{\mathit{fm}}\ h\ \bot = \bot \qquad \mathit{amap}_{\mathit{fm}}\ h\ (A\ a) = h\ a$
$\mathit{amap}_{\mathit{fm}}\ h\ (\varphi_1 \wedge \varphi_2) = \mathit{and}\ (\mathit{amap}_{\mathit{fm}}\ h\ \varphi_1)\ (\mathit{amap}_{\mathit{fm}}\ h\ \varphi_2)$
$\mathit{amap}_{\mathit{fm}}\ h\ (\varphi_1 \vee \varphi_2) = \mathit{or}\ (\mathit{amap}_{\mathit{fm}}\ h\ \varphi_1)\ (\mathit{amap}_{\mathit{fm}}\ h\ \varphi_2)$
$\mathit{amap}_{\mathit{fm}}\ h\ (\neg\ \varphi) = \mathit{neg}\ (\mathit{amap}_{\mathit{fm}}\ h\ \varphi)$

Both mapping functionals are only defined and needed for *qfree* formulae.

The set of atoms in a formula is computed by the (trivial to define) function $\mathit{atoms} :: \alpha\ \mathit{fm} \Rightarrow \alpha\ \mathit{set}$.

3.2 Interpretation

The interpretation or semantics of a *fm* is defined via the obvious homomorphic mapping to an HOL formula: $\wedge$ becomes $\wedge$, $\vee$ becomes $\vee$, etc. The interpretation of atoms is a parameter of this mapping. Atoms may refer to variables and are thus interpreted w.r.t. a valuation. Since variables are represented as natural numbers, the valuation is naturally represented as a list: variable i refers to the ith entry in the list (starting with 0). This leads to the following interpretation function $\mathit{interpret} :: (\alpha \Rightarrow \beta\ \mathit{list} \Rightarrow \mathit{bool}) \Rightarrow \alpha\ \mathit{fm} \Rightarrow \beta\ \mathit{list} \Rightarrow \mathit{bool}$:

$$\begin{array}{l}
\mathit{interpret}\ h\ \top\ \mathit{xs} = \mathit{True} \qquad \mathit{interpret}\ h\ \bot\ \mathit{xs} = \mathit{False} \\
\mathit{interpret}\ h\ (A\ a)\ \mathit{xs} = h\ a\ \mathit{xs} \\
\mathit{interpret}\ h\ (\varphi_1 \wedge \varphi_2)\ \mathit{xs} = (\mathit{interpret}\ h\ \varphi_1\ \mathit{xs} \wedge \mathit{interpret}\ h\ \varphi_2\ \mathit{xs}) \\
\mathit{interpret}\ h\ (\varphi_1 \vee \varphi_2)\ \mathit{xs} = (\mathit{interpret}\ h\ \varphi_1\ \mathit{xs} \vee \mathit{interpret}\ h\ \varphi_2\ \mathit{xs}) \\
\mathit{interpret}\ h\ (\neg\ \varphi)\ \mathit{xs} = (\neg\ \mathit{interpret}\ h\ \varphi\ \mathit{xs}) \\
\mathit{interpret}\ h\ (\exists\ \varphi)\ \mathit{xs} = (\exists x.\ \mathit{interpret}\ h\ \varphi\ (x \cdot \mathit{xs}))
\end{array}$$

In the equation for $\exists$ the value of the bound variable x is added at the front of the valuation. De Bruijn indexing ensures that in the body 0 refers to x and $i+1$ refers to bound variable i further up.

3.3 Atoms

Atoms are more than a type parameter α. They come with an *interpretation* (their semantics), and a few other specific functions. These functions are also parameters of the generic part of quantifier elimination. Thus the further development will be like a module parameterized with the type of atoms and some functions on atoms. These parameters will be instantiated later on when applying the framework to various linear arithmetics.

In Isabelle this parameterization is achieved by means of a **locale** [1], a named context of types, functions and assumptions about them. We call this context *ATOM*. It provides the following functions

$$\begin{array}{ll}
I_a & :: \alpha \Rightarrow \beta\ \mathit{list} \Rightarrow \mathit{bool} \\
\mathit{aneg} & :: \alpha \Rightarrow \alpha\ \mathit{fm} \\
\mathit{depends}_0 & :: \alpha \Rightarrow \mathit{bool} \\
\mathit{decr} & :: \alpha \Rightarrow \alpha
\end{array}$$

with the following intended meaning:

$I_a\ a\ \mathit{xs}$ is the interpretation of atom a w.r.t. valuation xs, where variable i (note $i :: \mathit{nat}$ because of de Bruijn) is assigned the ith element of xs.

aneg negates an atom. It returns a formula which should be free of negations. This is strictly for convenience: it means we can eliminate all negations from a formula. In the worst case we would have to introduce negated versions of all atoms, but in the case of linear orders this is not necessary because we can turn, for example, $\neg(x < y)$ into $(y < x) \vee (y = x)$.

$\mathit{depends}_0\ a$ checks if atom a contains (depends on) variable 0 and $\mathit{decr}\ a$ decrements every variable in a by 1.

Within context *ATOM* we introduce the abbreviation $I \equiv \mathit{interpret}\ I_a$. The assumptions on the parameters of *ATOM* can now be stated quite succinctly:

$$I\ (aneg\ a)\ xs = (\neg\ I_a\ a\ xs) \qquad nqfree\ (aneg\ a)$$
$$\neg\ depends_0\ a \Longrightarrow I_a\ a\ (x \cdot xs) = I_a\ (decr\ a)\ xs$$

Function *aneg* must return a quantifier and negation-free formula whose interpretation is the negation of the input. And when interpreting an atom not containing variable 0 we can drop the head of the valuation and decrement the variables without changing the interpretation.

These assumptions must be discharged when the locale is instantiated. We do not show this in the text because the proofs are straightforward in all cases.

In the context of *ATOM* we define two auxiliary functions: $atoms_0\ \varphi$ computes the list of all atoms in φ that depend on variable 0. The *negation normal form* (NNF) of a *qfree* formula is defined in the customary manner by pushing negations inwards. We show only a few representative equations:

$$\begin{aligned}
&nnf\ (\neg\ (A\ a)) = aneg\ a\\
&nnf\ (\varphi_1 \lor \varphi_2) = (nnf\ \varphi_1 \lor nnf\ \varphi_2)\\
&nnf\ (\neg\ (\varphi_1 \lor \varphi_2)) = (nnf\ (\neg\ \varphi_1) \land nnf\ (\neg\ \varphi_2))\\
&nnf\ (\neg\ (\varphi_1 \land \varphi_2)) = (nnf\ (\neg\ \varphi_1) \lor nnf\ (\neg\ \varphi_2))
\end{aligned}$$

The first equation differs from the usual definition and gets rid of negations altogether — see the explanation of *aneg* above.

3.4 Quantifier Elimination

The elimination of all quantifiers from a formula is achieved by eliminating them one by one in a bottom-up fashion. Thus each step needs to deal merely with the elimination of a single quantifier in front of a quantifier-free formula. This step is theory-dependent and hard. The lifting to arbitrary formulae is simple and can be done once and for all. We assume we are given a function $qe :: \alpha\ fm \Rightarrow \alpha\ fm$ for the elimination of a single $\exists$, i.e. $I\ (qe\ \varphi) = I\ (\exists\ \varphi)$ if $qfree\ \varphi$. Note that *qe* is not applied to $\exists\ \varphi$ but just to φ, $\exists$ remains implicit. Lifting *qe* is straightforward:

$$lift\text{-}nnf\text{-}qe :: (\alpha\ fm \Rightarrow \alpha\ fm) \Rightarrow \alpha\ fm \Rightarrow \alpha\ fm$$

$$\begin{aligned}
&lift\text{-}nnf\text{-}qe\ qe\ (\varphi_1 \land \varphi_2) = and\ (lift\text{-}nnf\text{-}qe\ qe\ \varphi_1)\ (lift\text{-}nnf\text{-}qe\ qe\ \varphi_2)\\
&lift\text{-}nnf\text{-}qe\ qe\ (\varphi_1 \lor \varphi_2) = or\ (lift\text{-}nnf\text{-}qe\ qe\ \varphi_1)\ (lift\text{-}nnf\text{-}qe\ qe\ \varphi_2)\\
&lift\text{-}nnf\text{-}qe\ qe\ (\neg\ \varphi) = neg\ (lift\text{-}nnf\text{-}qe\ qe\ \varphi)\\
&lift\text{-}nnf\text{-}qe\ qe\ (\exists\ \varphi) = qe\ (nnf\ (lift\text{-}nnf\text{-}qe\ qe\ \varphi))\\
&lift\text{-}nnf\text{-}qe\ qe\ \varphi = \varphi
\end{aligned}$$

Note that *qe* is called with an argument already in NNF. We can go even further and put the argument of *qe* into DNF. This is detailed elsewhere [15] but avoided here because it can lead to non-elementary complexity.

3.5 Correctness

Correctness *lift-nnf-qe* is roughly expressed as follows: if *qe* eliminates one existential while preserving the interpretation, then *lift-nnf-qe qe* eliminates all quantifiers while preserving the interpretation.

For compactness we employ a set theoretic language for expressing properties of functions: $A \to B$ is the set of functions from A to B and $|P| \equiv \{x \mid P\ x\}$.

Elimination of all quantifiers is easy:

Lemma 1. *If* $qe \in |nqfree| \to |qfree|$ *then* $qfree\ (lift\text{-}nnf\text{-}qe\ qe\ \varphi)$.

Preservation of the interpretation is slightly more involved:

Lemma 2. *If* $qe \in |nqfree| \to |qfree|$ *and for all* φ *and* xs: $(nqfree\ \varphi \Longrightarrow I\ (qe\ \varphi)\ xs = (\exists\, x.\ I\ \varphi\ (x \cdot xs)))$, *then* $I\ (lift\text{-}nnf\text{-}qe\ qe\ \varphi)\ xs = I\ \varphi\ xs$.

In the following sections we define a number of quantifier elimination functions called f_1 (for different names f) that eliminate a single $\exists$. In each case we have proved that f_1 satisfies the assumptions of the above two lemmas (with f_1 for qe), define $f = lift\text{-}nnf\text{-}qe\ f_1$ and thus obtain $qfree\ (f\ \varphi)$ and $I\ (f\ \varphi)\ xs = I\ \varphi\ xs$ as corollaries. Because of this uniformity and because the correctness proofs are either discussed informally beforehand or are well-known from the literature, we suppress all of this in the presentation. Thus it may look as if we merely present code, but the proofs are all there!

4 Dense Linear Orders

The theory of dense linear orders (without endpoints) is an extension of the theory of linear orders with the axioms

$$x < z \Longrightarrow \exists\, y.\ x < y \wedge y < z \qquad \exists\, u.\ x < u \qquad \exists\, l.\ l < x$$

It is the canonical example of quantifier elimination [11]. The equivalence $(\exists\, y.\ x < y \wedge y < z) = (x < z)$ is an easy consequence of the axioms and the essence of Fourier's elimination method, which requires conversion to DNF and is thus non-elementary.

In contrast we develop two new NNF-based algorithms based on the *test point* method (originally due to Cooper [5] and Ferrante and Rackoff [6] and later generalized by Weispfenning [19]). The idea is to find a finite set of test points T (depending on φ) such that $(\exists x.\ \varphi(x)) = (\bigvee_{t \in T} \varphi(t))$. The complication is that (conceptually) T may contain values like infinity, infinitesimals or intermediate points, values that are not representable in the given term language. The challenge is to define special versions of substitution for these values.

4.1 Atoms

There are just the two relations $<$ and $=$ and no function symbols. Thus atomic formulae can be represented by the following datatype:

$$\textbf{datatype}\ atom = nat\ \boldsymbol{<}\ nat \mid nat\ \boldsymbol{=}\ nat$$

Note the **bold** infix constructors $\boldsymbol{<}$ and $\boldsymbol{=}$. Because there are no function symbols, the arguments of the relations must be variables. For example, $i\ \boldsymbol{<}\ j$ represents the atom $x_i < x_j$ in de Bruijn notation.

Now we can instantiate locale *ATOM*. Type parameter α becomes type *atom*. The interpretation function I_a becomes I_{dlo} where

$$I_{dlo}\ (i = j)\ xs = (xs_{[i]} = xs_{[j]}) \qquad I_{dlo}\ (i < j)\ xs = (xs_{[i]} < xs_{[j]})$$

The notation $xs_{[i]}$ means selection of the ith element of xs. The type of I_{dlo} is explicitly restricted such that xs must be a list of elements over a dense linear order, where the latter is formalized as a type class [8] with the axioms shown at the start of this section. Thus all valuations in this section are over dense linear orders. Parameter *aneg* becomes neg_{dlo}:

$$neg_{dlo}\ (i < j) = (A\ (j < i) \lor A\ (i = j))$$
$$neg_{dlo}\ (i = j) = (A\ (i < j) \lor A\ (j < i))$$

The parameters *adepends* and *adecr* are instantiated with $depends_{dlo}$ and $decr_{dlo}$:

$$depends_{dlo}\ (i = j) = (i = 0 \lor j = 0)$$
$$depends_{dlo}\ (i < j) = (i = 0 \lor j = 0)$$

$$decr_{dlo}\ (i < j) = (i - 1 < j - 1) \qquad decr_{dlo}\ (i = j) = (i - 1 = j - 1)$$

This instantiation satisfies all the axioms of *ATOM*.

4.2 The Interior Point Method

Ferrante and Rackoff [6] realized (for linear real arithmetic) that when eliminating x from ϕ it (essentially) suffices to collect all lower bounds l of x (i.e. $l < x$ occurs in ϕ) and all upper bounds u of x (i.e. $x < u$ occurs in ϕ) and try all such $(l + u)/2$ as test points. This method is implemented in §5.2.

Now we present a novel quantifier elimination method for DLO based on Ferrante and Rackoff's idea. The problem with DLO is that one cannot name any point between two variables x and y. Hence a special form of substitution must be defined that behaves as if some intermediate point was substituted without requiring such a point. We use the symbolic notation $x{\downarrow}y$ to denote some arbitrary but fixed point in the interval (x, y). The key cases in defining substitution with $x{\downarrow}y$ are: $(x{\downarrow}y < z) = (y \leq z)$, $(z < x{\downarrow}y) = (z \leq x)$, $(x{\downarrow}y < x{\downarrow}y) = False$, $(x{\downarrow}y = x{\downarrow}y) = True$ and $(x{\downarrow}y = z) = False$. The last equation is motivated because we can always choose $x{\downarrow}y$ to be different from z. Note also that these definitions only work as expected if $x < y$.

We also need the fictitious values $-\infty$ and ∞ first used by Cooper. Then we can formulate the interior point method as a logical equivalence in test point form, where ϕ must be quantifier-free and in NNF:

$$(\exists x.\ \phi(x)) \ = \ (\phi(-\infty) \lor \phi(\infty) \lor \bigvee_{y \in E} \phi(y) \lor \bigvee_{y \in L, z \in U} (y < z \land \phi(y{\downarrow}z))) \qquad (1)$$

E is the set of y such that $x = y$ or $y = x$ occur in $\phi(x)$, L is the set of y such that $y < x$ occurs in $\phi(x)$, U is the set of y such that $x < y$ occurs in $\phi(x)$, where x is the bound variable and y is different from x.

We sketch a proof of (1), details can be found in the Isabelle proof. The if-direction is easy as in each case a witness is given. Except that $-\infty$, ∞ and $y{\downarrow}z$ are not proper values. But by induction on ϕ one can show that $\phi(-\infty)$ etc imply $\phi(x)$ for suitable x:

$$\exists x.\forall y \le x.\ \phi(-\infty) = \phi(y) \qquad \exists x.\forall y \ge x.\ \phi(\infty) = \phi(y)$$
$$y < z \wedge \phi(y{\downarrow}z) \Longrightarrow \forall x \in (y,z).\ \phi(x)$$

For the only-if-direction assume $\phi(x)$ and not $\phi(-\infty) \vee \phi(\infty) \vee \bigvee_{y\in E} \phi(y)$. We have to show that $\phi(y{\downarrow}z)$ for some $y \in L$ and $z \in U$. From the assumptions it follows by induction on ϕ that there must be $y_0 \in L$ and $z_0 \in U$ such that $x \in (y_0, z_0)$. Now we show (by induction on ϕ) the lemma that innermost intervals (y, z) completely satisfy ϕ:

Lemma 3. *If $x \in (y,z)$, $x \notin E$, $(y,x) \cap L = \emptyset$ and $(x,z) \cap U = \emptyset$, then $\phi(x)$ implies $\forall u \in (y,z).\ \phi(u)$.*

Given $x \in (y_0, z_0)$ we define $y = \max\{y \in L \mid y < x\}$ and $z = \min\{z \in U \mid x < z\}$. It is easy to see that this satisfies the premises of the lemma and hence $\forall u \in (y,z).\ \phi(u)$. Again by induction on ϕ one can show that this actually implies $\phi(y{\downarrow}z)$:

Lemma 4. *If $x \in (y,z)$, $x \notin E$, $(y,x) \cap L = \emptyset$ and $(x,z) \cap U = \emptyset$, then $(\forall x \in (y,z).\ \phi(x))$ implies $\phi(y{\downarrow}z)$.*

4.3 A Verified Implementation of the Interior Point Method

The executable version of (1) is short but requires some auxiliary functions.

```
$interior_1\ \varphi =$
$(let\ as = atoms_0\ \varphi;\ lbs = lbounds\ as;\ ubs = ubounds\ as;\ ebs = ebounds\ as;$
    $intrs = [\ A(l < u) \wedge (subst_2\ l\ u\ \varphi).\ l{\leftarrow}lbs,\ u{\leftarrow}ubs]$
 $in\ list\text{-}disj\ (inf_-\ \varphi \cdot inf_+\ \varphi \cdot intrs\ @\ map\ (subst\ \varphi)\ ebs))$
```

We will now explain the ingredients.

The implementation of substituting $l{\downarrow}u$ in atoms is given below. Please note that substitution must not just substitute for variable 0 but must also decrement the other variables.

$asubst_2\ l\ u\ (0 < 0) = \bot \qquad asubst_2\ l\ u\ (Suc\ i < Suc\ j) = A\ (i < j)$
$asubst_2\ l\ u\ (0 < Suc\ j) = (A\ (u < j) \vee A\ (u = j))$
$asubst_2\ l\ u\ (Suc\ i < 0) = (A\ (i < l) \vee A\ (i = l))$
$asubst_2\ l\ u\ (0 = 0) = \top \qquad asubst_2\ l\ u\ (Suc\ i = Suc\ j) = A\ (i = j)$
$asubst_2\ l\ u\ (0 = Suc\ v) = \bot \qquad asubst_2\ l\ u\ (Suc\ v = 0) = \bot$

From atoms to formulae is a short step: $subst_2\ l\ u\ \varphi \equiv amap_{fm}\ (asubst_2\ l\ u)\ \varphi$

Plain old substitution of one variable for 0 is defined first on variables, then on atoms and finally on formulae:

$\mathit{isubst}\ k\ \mathit{0} = k \quad \mathit{isubst}\ k\ (\mathit{Suc}\ i) = i$

$\mathit{asubst}\ k\ (i < j) = (\mathit{isubst}\ k\ i < \mathit{isubst}\ k\ j)$
$\mathit{asubst}\ k\ (i = j) = (\mathit{isubst}\ k\ i = \mathit{isubst}\ k\ j)$

$\mathit{subst}\ \varphi\ k \equiv \mathit{map}_{\mathit{fm}}\ (\mathit{asubst}\ k)\ \varphi$

Substituting $-\infty$ for 0 is implemented as follows:

$\mathit{amin\text{-}inf}\ (i < \mathit{0}) = \bot \qquad \mathit{amin\text{-}inf}\ (\mathit{0} < \mathit{Suc}\ j) = \top$
$\mathit{amin\text{-}inf}\ (\mathit{Suc}\ i < \mathit{Suc}\ j) = A\ (i < j)$
$\mathit{amin\text{-}inf}\ (\mathit{0} = \mathit{0}) = \top \qquad \mathit{amin\text{-}inf}\ (\mathit{Suc}\ i = \mathit{Suc}\ j) = A\ (i = j)$
$\mathit{amin\text{-}inf}\ (\mathit{0} = \mathit{Suc}\ v) = \bot \qquad \mathit{amin\text{-}inf}\ (\mathit{Suc}\ v = \mathit{0}) = \bot$
$\mathit{inf}_-\ \varphi \equiv \mathit{amap}_{\mathit{fm}}\ \mathit{amin\text{-}inf}\ \varphi$

Dually there is inf_+ for substituting ∞. Lower bounds, upper bounds and equalities are conveniently collected from a list of atoms by list comprehension:

$\mathit{lbounds}\ \mathit{as} = [i.\ (\mathit{Suc}\ i < \mathit{0}) \leftarrow \mathit{as}] \quad \mathit{ubounds}\ \mathit{as} = [i.\ (\mathit{0} < \mathit{Suc}\ i) \leftarrow \mathit{as}]$
$\mathit{ebounds}\ \mathit{as} = [i.\ (\mathit{Suc}\ i = \mathit{0}) \leftarrow \mathit{as}]\ @\ [i.\ (\mathit{0} = \mathit{Suc}\ i) \leftarrow \mathit{as}]$

4.4 The Method of Infinitesimals

Loos and Weispfenning [12] proposed a quantifier elimination procedure for linear real arithmetic (see §5.3) where test points are $x + \varepsilon$ (for x a lower bound) or $y - \varepsilon$ (for y an upper bound) where ε is an infinitesimal. That is, the test points are arbitrarily close to the lower or upper bounds of the eliminated variable. In particular, it is not necessary to pair all lower and upper bounds but one can choose either set, typically the smaller one. For succinctness we ignore this duality and concentrate on the lower bounds only.

In this section we adapt the idea of infinitesimals to derive a new quantifier elimination procedure for DLO. We merely need to explain what substitution of $x+\varepsilon$ means: $(x+\varepsilon < y) = (x < y)$, $(y < x+\varepsilon) = (y \le x)$, $(x+\varepsilon < x+\varepsilon) = \mathit{False}$, $(x+\varepsilon = x+\varepsilon) = \mathit{True}$, $(x+\varepsilon = y) = \mathit{False}$, where x and y are different variables.

The test point method with infinitesimals is justified by the following equivalence, where, as usual, ϕ is quantifier free and in NNF:

$$(\exists x.\ \phi(x)) \ = \ (\phi(-\infty) \vee \bigvee_{y \in E} \phi(y) \vee \bigvee_{y \in L} \phi(y+\varepsilon)) \tag{2}$$

where E and L are defined as in (1). The proof is also similar. The main differences are: For the if-direction we need to show (by induction on ϕ) that $y + \varepsilon$ represents a proper witness:

$$\phi(y+\varepsilon) \Longrightarrow \exists y' > y. \forall x \in (y, y').\ \phi(x)$$

The two lemmas for the only-if-direction become

Lemma 5. *If $y < x$, $x \notin E$, $(y, x) \cap L = \emptyset$ and $\phi(x)$, then $\forall u \in (y, x].\ \phi(u)$.*

Lemma 6. *If $y < x$, $x \notin E$, $(y, x) \cap L = \emptyset$ and $\forall u \in (y, x]$. $\phi(u)$, then $\phi(y + \varepsilon)$.*

Our verified implementation of ([2])

$$\begin{array}{l} \mathit{eps}_1\ \varphi = (\mathsf{let}\ \mathit{as} = \mathit{atoms}_0\ \varphi;\ \mathit{lbs} = \mathit{lbounds}\ \mathit{as};\ \mathit{ebs} = \mathit{ebounds}\ \mathit{as} \\ \qquad\qquad \mathsf{in}\ \mathit{list\text{-}disj}\ (\mathit{inf}_-\ \varphi\ \cdot\ \mathit{map}\ (\mathit{subst}_+\ \varphi)\ \mathit{lbs}\ @\ \mathit{map}\ (\mathit{subst}\ \varphi)\ \mathit{ebs})) \end{array}$$

requires only one new concept, $\mathit{subst}_+\ \varphi\ y$, the substitution $\phi(y + \varepsilon)$:

$$\begin{array}{ll} \mathit{asubst}_+\ k\ (0 < 0) = \bot & \mathit{asubst}_+\ k\ (\mathit{Suc}\ i < \mathit{Suc}\ j) = A\ (i < j) \\ \mathit{asubst}_+\ k\ (0 < \mathit{Suc}\ j) = A\ (k < j) & \\ \multicolumn{2}{l}{\mathit{asubst}_+\ k\ (\mathit{Suc}\ i < 0) = (\mathsf{if}\ i = k\ \mathsf{then}\ \top\ \mathsf{else}\ A\ (i < k) \vee A\ (i = k))} \\ \mathit{asubst}_+\ k\ (0 = 0) = \top & \mathit{asubst}_+\ k\ (\mathit{Suc}\ i = \mathit{Suc}\ j) = A\ (i = j) \\ \mathit{asubst}_+\ k\ (0 = \mathit{Suc}\ v) = \bot & \mathit{asubst}_+\ k\ (\mathit{Suc}\ v = 0) = \bot \end{array}$$

$$\mathit{subst}_+\ \varphi\ k \equiv \mathit{amap}_{\mathit{fm}}\ (\mathit{asubst}_+\ k)\ \varphi$$

4.5 Complexity

A formula of size n can contain at most n variables. The set of variables decreases by one in each step. In the worst case all of them are bound and need to be eliminated. In each step of the quantifier elimination processes ([1]) and ([2]) the sets E, L and U are at most as large as k, the current number of variables.

The interior point method makes at most $(k-1)^2$ copies of the formula in each step. Hence the size of the output formula and also the amount of working space required is $O(n \cdot (n-1)^2 \cdots 1^2) = O(n \cdot (n-1)!^2)$. The method of infinitesimals, however, only makes at most $k-1$ copies, thus requiring only $O(n \cdot (n-1) \cdots 1) = O(n!)$ space. The time complexity of both algorithms is linear in their space complexity, i.e. time and space coincide.

5 Linear Real Arithmetic

Linear real arithmetic is concerned with terms built up from variables, constants, addition, and multiplication with constants. Relations between such terms can be put into a normal form $r \bowtie c_0 * x_0 + \cdots c_n * x_n$ with $\bowtie \in \{=, <\}$ and $r, c_0, \ldots, c_n \in \mathbb{R}$. It is this normal form we work with in this section.

Note that although we phrase everything in terms of the real numbers, the rational numbers work just as well. In fact, any ordered, divisible, torsion free, Abelian group will do.

We present verified implementations of two quantifier elimination procedures: one due to Ferrante and Rackoff [6] and one due to Loos and Weispfenning [12].

5.1 Atoms

Type *atom* formalizes the normal forms explained above:

$$\textbf{datatype}\ \mathit{atom} = \mathit{real} < (\mathit{real\ list})\ \mid\ \mathit{real} = (\mathit{real\ list})$$

The second constructor argument is the list of coefficients $[c_0,\ldots,c_n]$ of the variables *0* to *n* — remember de Bruijn! Coefficient lists should be viewed as vectors and we define the usual vector operations on them:

$x *_s xs$ is the componentwise multiplication of a scalar x with a vector xs.
$xs + ys$ and $xs - ys$ are componentwise addition and subtraction of vectors.
$\langle xs,ys\rangle = (\sum (x,y) \leftarrow zip\ xs\ ys.\ x{*}y)$ is the inner product of two vectors, i.e. the sum over the componentwise products.

If the two vectors involved in an operation are of different length, the shorter one is padded with 0s (as in Obua's treatment of matrices [18]). We can prove all the algebraic properties we need, like $\langle xs + ys,zs\rangle = \langle xs,zs\rangle + \langle ys,zs\rangle$.

Now we instantiate locale *ATOM* just like for DLO in §4.1. The main function is the interpretation I_R of atoms, which is straightforward:

$$I_R\ (r < cs)\ xs = (r < \langle cs,xs\rangle) \qquad I_R\ (r = cs)\ xs = (r = \langle cs,xs\rangle)$$

5.2 Ferrante and Rackoff

Ferrante and Rackoff [6], inspired by Cooper [5], avoided DNF conversions by the test point method explained in §4. We have already explained the key idea of Ferrante and Rackoff in §4.2. If you replace $y{\downarrow}z$ in (1) by $(y+z)/2$ you almost obtain their algorithm. In principle any point between y and z works but $(y+z)/2$ also takes care of equalities: they lump E, L and U together (to be avoided in an implementation) but because $(y+y)/2 = y$ this recovers E. As their algorithm is well-known, we present its optimized and verified implementation right away:

$FR_1\ \varphi =$
$(\mathsf{let}\ as = atoms_0\ \varphi;\ lbs = lbounds\ as;\ ubs = ubounds\ as;\ ebs = ebounds\ \varphi;$
$\quad intrs = [subst\ \varphi\ (between\ l\ u)\ .\ l \leftarrow lbs,\ u \leftarrow ubs];$
$\ \mathsf{in}\ list\text{-}disj\ (inf_-\ \varphi\ \cdot\ inf_+\ \varphi\ \cdot\ intrs\ @\ map\ (subst\ \varphi)\ ebs))$

Except for the definition of *intrs* this looks identical to the definition of $interior_1$ in §4.3. However, all auxiliary functions are different: they operate on pairs (r, cs) which, under a valuation xs, represent the value $r + \langle cs,xs\rangle$. First the various bounds are extracted:

$lbounds\ as = [(r/c,\ (-1/c) *_s cs).\ (r < (c \cdot cs)) \leftarrow as,\ c{>}0]$
$ubounds\ as = [(r/c,\ (-1/c) *_s cs).\ (r < (c \cdot cs)) \leftarrow as,\ c{<}0]$
$ebounds\ as = [(r/c,\ (-1/c) *_s cs).\ (r = (c \cdot cs)) \leftarrow as,\ c{\neq}0]$

The intermediate point between two such points is easy:

$$between\ (r,\ cs)\ (s,\ ds) = ((r + s)\ /\ 2,\ (1\ /\ 2) *_s (cs + ds))$$

We need both ordinary substitution of $(r,\ cs)$ pairs

$asubst\ (r,\ cs)\ (s < d \cdot ds) = (s - d * r < d *_s cs + ds)$
$asubst\ (r,\ cs)\ (s = d \cdot ds) = (s - d * r = d *_s cs + ds)$
$asubst\ rcs\ a = a$

$subst\ \varphi\ rcs \equiv map_{fm}\ (asubst\ rcs)\ \varphi$

and substitution inf_- of $-\infty$ (and the analogous version inf_+ for ∞):

$$\begin{aligned}
&\mathit{inf}_-\ (\varphi_1 \wedge \varphi_2) = \mathit{and}\ (\mathit{inf}_-\ \varphi_1)\ (\mathit{inf}_-\ \varphi_2)\\
&\mathit{inf}_-\ (\varphi_1 \vee \varphi_2) = \mathit{or}\ (\mathit{inf}_-\ \varphi_1)\ (\mathit{inf}_-\ \varphi_2)\\
&\mathit{inf}_-\ (A\ (r < c \cdot cs)) = (\textsf{if}\ c < 0\ \textsf{then}\ \top\ \textsf{else if}\ 0 < c\ \textsf{then}\ \bot\ \textsf{else}\ A\ (r < cs))\\
&\mathit{inf}_-\ (A\ (r = c \cdot cs)) = (\textsf{if}\ c = 0\ \textsf{then}\ A\ (r = cs)\ \textsf{else}\ \bot)
\end{aligned}$$

The remaining cases are the identity. This concludes the auxiliary functions.

5.3 Loos and Weispfenning

The method of infinitesimals described in §4.4 was inspired by the analogous method for linear real arithmetic proposed by Loos and Weispfenning [12] who also showed practical examples where it outperforms Ferrante and Rackoff. Yet this method seems relatively unknown in the literature. Its implementation eps_1 is textually identical to the one for DLO in §4.4. But the auxiliary functions differ. Luckily we have seen all of them already, except subst_+:

$$\begin{aligned}
&\mathit{asubst}_+\ (r,\ cs)\ (s < d \cdot ds) =\\
&(\textsf{if}\ d = 0\ \textsf{then}\ A\ (s < ds)\\
&\ \textsf{else let}\ u = s - d * r;\ v = d *_s cs + ds;\ \mathit{lessa} = A\ (u < v)\\
&\qquad \textsf{in if}\ d < 0\ \textsf{then}\ \mathit{lessa}\ \textsf{else}\ \mathit{lessa} \vee A\ (u = v))\\
&\mathit{asubst}_+\ \mathit{rcs}\ (r = d \cdot ds) = (\textsf{if}\ d = 0\ \textsf{then}\ A\ (r = ds)\ \textsf{else}\ \bot)\\
&\mathit{asubst}_+\ \mathit{rcs}\ a = A\ a
\end{aligned}$$

$$\mathit{subst}_+\ \varphi\ \mathit{rcs} \equiv \mathit{amap}_{\mathit{fm}}\ (\mathit{asubst}_+\ \mathit{rcs})\ \varphi$$

6 Presburger Arithmetic

Presburger arithmetic needs a divisibility (or congruence) predicate "|" to allow quantifier elimination. On the other hand we restrict our attention to $\leq$ because $i < j$ is equivalent with $i + 1 \leq j$. Thus all atoms are of the form $i \leq k_0 * x_0 + \cdots + k_n * x_n$ or $d \parallel i + k_0 * x_0 + \cdots k_n * x_n$, where $\parallel$ is $|$ or $\nmid$, and $d, i, k_0, \ldots, k_n \in \mathbb{Z}$ and $d > 0$. This becomes the **datatype**

$$\mathit{atom} = \mathit{Le}\ \mathit{int}\ (\mathit{int\ list})\ |\ \mathit{Dvd}\ \mathit{int}\ \mathit{int}\ (\mathit{int\ list})\ |\ \mathit{NDvd}\ \mathit{int}\ \mathit{int}\ (\mathit{int\ list})$$

We have avoided infix constructors because they work less well for ternary operations. Atoms are interpreted w.r.t. a list of variables as usual:

$$\begin{aligned}
&I_Z\ (\mathit{Le}\ i\ ks)\ xs = (i \leq \langle ks,xs\rangle)\\
&I_Z\ (\mathit{Dvd}\ d\ i\ ks)\ xs = d \mid (i + \langle ks,xs\rangle)\\
&I_Z\ (\mathit{NDvd}\ d\ i\ ks)\ xs = (\neg\ d \mid (i + \langle ks,xs\rangle))
\end{aligned}$$

Note that we reuse the polymorphic vector, i.e. list operations like $\langle .,. \rangle$ introduced for linear real arithmetic: they are defined for arbitrary types with 0, + and $*$.

The parameters of locale $ATOM$ are instantiated as follows. The interpretation of atoms is given by function I_Z above, their negation by

$$neg_Z\ (Le\ i\ ks) = A\ (Le\ (1\ -\ i)\ (-\ ks))$$
$$neg_Z\ (Dvd\ d\ i\ ks) = A\ (NDvd\ d\ i\ ks) \qquad neg_Z\ (NDvd\ d\ i\ ks) = A\ (Dvd\ d\ i\ ks)$$

and their decrementation by

$$decr_Z\ (Le\ i\ ks) = Le\ i\ (tl\ ks)$$
$$decr_Z\ (Dvd\ d\ i\ ks) = Dvd\ d\ i\ (tl\ ks) \quad decr_Z\ (NDvd\ d\ i\ ks) = NDvd\ d\ i\ (tl\ ks)$$

Parameter $depends_0$ becomes $\lambda a.\ hd\text{-}coeff\ a \neq 0$ where

$$hd\text{-}coeff\ (Le\ i\ ks) = (\mathsf{case}\ ks\ \mathsf{of}\ [] \Rightarrow 0\ |\ k \cdot x \Rightarrow k)$$
$$hd\text{-}coeff\ (Dvd\ d\ i\ ks) = (\mathsf{case}\ ks\ \mathsf{of}\ [] \Rightarrow 0\ |\ k \cdot x \Rightarrow k)$$
$$hd\text{-}coeff\ (NDvd\ d\ i\ ks) = (\mathsf{case}\ ks\ \mathsf{of}\ [] \Rightarrow 0\ |\ k \cdot x \Rightarrow k)$$

6.1 Cooper's Algorithm

Cooper's algorithm relies on Cooper's theorem [5] which holds provided all coefficients of x in $\phi(x)$ are 1 or -1 (or 0):

$$(\exists x.\ \phi(x))\ =\ (\bigvee_{j\in(0,\delta-1)} \phi_{-\infty}(j) \vee \bigvee_{y\in L} \bigvee_{j\in(0,\delta-1)} \phi(y+j))$$

where δ is the lcm of all d such that $d \mid t$ or $d \nmid t$ occurs in $\phi(x)$ and t contains x, L is the set of lower bounds for x in $\phi(x)$, and $\phi_{-\infty}(j)$ is $\phi(x)$ where x has been replaced by $-\infty$ in all inequations and by j in all other atoms.

We start by setting all (non-zero) head coefficients to 1 or -1. This is achieved by multiplying each atom a (with non-zero head coefficient) with m/k, where m is the lcm of all (non-zero) head coefficients and k is a's head coefficient (assume $k > 0$ for simplicity). Now all (non-zero) head coefficients are m, we replace them by 1 and conjoin the atom $m \mid x_0$. This is what $hd\text{-}coeff1$ does for an atom and $hd\text{-}coeff1$ for a formula:

```
$hd\text{-}coeff1\ m\ (Le\ i\ (k \cdot ks)) =$
$(\mathsf{if}\ k = 0\ \mathsf{then}\ Le\ i\ (k \cdot ks)$
 $\mathsf{else\ let}\ m' = m\ div\ |k|\ \mathsf{in}\ Le\ (m' * i)\ (sgn\ k \cdot m' *_s ks))$
$hd\text{-}coeff1\ m\ (Dvd\ d\ i\ (k \cdot ks)) =$
$(\mathsf{if}\ k = 0\ \mathsf{then}\ Dvd\ d\ i\ (k \cdot ks)$
 $\mathsf{else\ let}\ m' = m\ div\ k\ \mathsf{in}\ Dvd\ (m' * d)\ (m' * i)\ (1 \cdot m' *_s ks))$
$hd\text{-}coeff1\ m\ (NDvd\ d\ i\ (k \cdot ks)) =$
$(\mathsf{if}\ k = 0\ \mathsf{then}\ NDvd\ d\ i\ (k \cdot ks)$
 $\mathsf{else\ let}\ m' = m\ div\ k\ \mathsf{in}\ NDvd\ (m' * d)\ (m' * i)\ (1 \cdot m' *_s ks))$
$hd\text{-}coeff1\ m\ a = a$

$hd\text{-}coeffs1\ \varphi =$
$(\mathsf{let}\ m = zlcms\ (map\ hd\text{-}coeff\ (atoms_0\ \varphi))$
 $\mathsf{in}\ A\ (Dvd\ m\ 0\ [1]) \wedge map_{fm}\ (hd\text{-}coeff1\ m)\ \varphi)$
```

The sign function *sgn* returns -1, 0, and 1 for negative, zero and positive arguments. Functions *zlcms* computes the positive lcm of a list of integers.

Now we start to implement Cooper's theorem. The substitution $\phi_{-\infty}(j)$ is implemented by the composition of

$inf_{-}\ (\varphi_1 \wedge \varphi_2) = and\ (inf_{-}\ \varphi_1)\ (inf_{-}\ \varphi_2)$
$inf_{-}\ (\varphi_1 \vee \varphi_2) = or\ (inf_{-}\ \varphi_1)\ (inf_{-}\ \varphi_2)$
$inf_{-}\ (A\ (Le\ i\ (k \cdot ks))) =$
$(\mathsf{if}\ k < 0\ \mathsf{then}\ \top\ \mathsf{else\ if}\ 0 < k\ \mathsf{then}\ \bot\ \mathsf{else}\ A\ (Le\ i\ (0 \cdot ks)))$
$inf_{-}\ \varphi = \varphi$

and ordinary substitution:

$asubst\ i'\ ks'\ (Le\ i\ (k \cdot ks)) = Le\ (i - k * i')\ (k *_s ks' + ks)$
$asubst\ i'\ ks'\ (Dvd\ d\ i\ (k \cdot ks)) = Dvd\ d\ (i + k * i')\ (k *_s ks' + ks)$
$asubst\ i'\ ks'\ (NDvd\ d\ i\ (k \cdot ks)) = NDvd\ d\ (i + k * i')\ (k *_s ks' + ks)$
$asubst\ i'\ ks'\ a = a$

$subst\ i\ ks\ \varphi \equiv map_{fm}\ (asubst\ i\ ks)\ \varphi$

The right-hand side of Cooper's theorem now becomes executable:

```
$cooper_1\ \varphi =$
$(\mathsf{let}\ as = atoms_0\ \varphi;\ d = zlcms(map\ divisor\ as);$
    $lbs = [(i,ks).\ Le\ i\ (k \cdot ks) \leftarrow as,\ k{>}0]$
 $\mathsf{in}\ or\ (Disj\ [0..d - 1]\ (\lambda n.\ subst\ n\ []\ (inf_{-}\ \varphi)))$
      $(Disj\ lbs\ (\lambda(i,ks).\ Disj\ [0..d - 1]\ (\lambda n.\ subst\ (i + n)\ (-ks)\ \varphi))))$
```

where $divisor\ (Dvd\ d\ \_\ \_) = divisor(NDvd\ d\ \_\ \_) = d$, $divisor\ (Le\ \_\ \_) = 1$ and $Disj\ us\ f \equiv list\text{-}disj\ (map\ f\ us)$. The lower bounds *lbs* are computed directly rather than by an auxiliary function.

The two phases of Cooper's algorithm are simply composed and lifted:

$cooper = lift\text{-}nnf\text{-}qe\ (cooper_1 \circ hd\text{-}coeffs1)$

6.2 Correctness

There is a slight complication we have glossed over so far. We want to exclude the atoms *Dvd 0 i ks* and *NDvd 0 i ks* because they behave anomalously and the algorithm does not generate them either. Catering for them would complicate the algorithm with case distinctions. In order to restrict attention to a subset of *normal* atoms, locale *ATOM* in fact has another parameter not mentioned so far: $anormal :: \alpha \Rightarrow bool$ with the axioms

$$anormal\ a \Longrightarrow \forall\, b{\in}atoms\ (aneg\ a).\ anormal\ b$$
$$\neg\ depends_0\ a \Longrightarrow anormal\ a \Longrightarrow anormal\ (decr\ a)$$

In words: negation and decrementation do not lead outside the normal atoms. These axioms allow to show the following refined version of Lemma 2 (inside *ATOM*), where $normal\ \varphi = (\forall\, a{\in}atoms\ \varphi.\ anormal\ a)$:

Lemma 7. *If* $qe \in |nqfree| \rightarrow |qfree|$ *and* $qe \in |nqfree| \cap |normal| \rightarrow |normal|$ *and for all* φ *and* xs*:* $normal\ \varphi \wedge nqfree\ \varphi \Longrightarrow I\ (qe\ \varphi)\ xs = (\exists\, x.\ I\ \varphi\ (x \cdot xs))$*, then* $normal\ \varphi$ *implies* $I\ (lift\text{-}nnf\text{-}qe\ qe\ \varphi)\ xs = I\ \varphi\ xs$*.*

In the instantiation of *ATOM* for Presburger arithmetic parameter *anormal* becomes $\lambda a.\ divisor\ a \neq 0$. The above lemma is instantiated with $cooper_1 \circ$ *hd-coeffs1* for *qe* and its premises are discharged by the detailed but familiar correctness arguments for Cooper's algorithm. We obtain the corollary $normal\ \varphi \Longrightarrow I\ (cooper\ \varphi)\ xs = I\ \varphi\ xs$. Of course $qfree\ (cooper\ \varphi)$ is also proved.

7 Related Work

The literature on decision procedures for linear arithmetic is vast. We concentrate on formally verified algorithms.

Nipkow [15] presents the generic framework of §3 in detail but concentrates on non-elementary DNF-based procedures. Chaieb and Nipkow [4] present a reflective implementations of Cooper's algorithm. But they lack the generic framework and they use special purpose data structures for terms instead of relying on lists as we do. As a result some of their functions are considerably more complicated than ours and theorems and proofs are littered with linearity assumptions that are implicit in our list representation. Hence they can only present part of their implementation. Chaieb [3] presents a verified combination of Ferrante-Rackoff and Cooper. Norrish [17] was the first to implement a proof-producing version of Cooper's algorithm in a theorem prover. Similar implementation of QE for complex numbers and for real closed fields are reported by Harrison [10] and McLaughlin [14]. The CAD QE procedure for real closed fields has been reflected but only partly verified by Mahboubi [13] in Coq.

Acknowledgment. Amine Chaieb alerted me to the infinitesimal approach [12]. Discussions with him and Jeremy Avigad were very helpful.

References

1. Ballarin, C.: Interpretation of locales in Isabelle: Theories and proof contexts. In: Borwein, J.M., Farmer, W.M. (eds.) MKM 2006. LNCS (LNAI), vol. 4108, pp. 31–43. Springer, Heidelberg (2006)
2. Boyer, R.S., Moore, J.S.: Metafunctions: proving them correct and using them efficiently as new proof procedures. In: Boyer, R., Moore, J. (eds.) The Correctness Problem in Computer Science, pp. 103–184. Academic Press, London (1981)
3. Chaieb, A.: Verifying mixed real-integer quantifier elimination. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 528–540. Springer, Heidelberg (2006)
4. Chaieb, A., Nipkow, T.: Verifying and reflecting quantifier elimination for Presburger arithmetic. In: Sutcliffe, G., Voronkov, A. (eds.) LPAR 2005. LNCS (LNAI), vol. 3835, pp. 367–380. Springer, Heidelberg (2005)

5. Cooper, D.C.: Theorem proving in arithmetic without multiplication. In: Meltzer, B., Michie, D. (eds.) Machine Intelligence, vol. 7, pp. 91–100. Edinburgh University Press (1972)
6. Ferrante, J., Rackoff, C.: A decision procedure for the first order theory of real addition with order. SIAM J. Computing 4, 69–76 (1975)
7. Gonthier, G.: A computer-checked proof of the four-colour theorem, http://research.microsoft.com/~gonthier/4colproof.pdf
8. Haftmann, F., Wenzel, M.: Constructive type classes in Isabelle. In: Altenkirch, T., McBride, C. (eds.) TYPES 2006. LNCS, vol. 4502, pp. 160–174. Springer, Heidelberg (2007)
9. Harrison, J.: Introduction to Logic and Automated Theorem Proving. Cambridge University Press, Cambridge (forthcoming)
10. Harrison, J.: Complex quantifier elimination in HOL. In: Boulton, R.J., Jackson, P.B. (eds.) TPHOLs 2001. LNCS, vol. 2152, pp. 159–174. Springer, Heidelberg (2001)
11. Langford, C.: Some theorems on deducibility. Annals of Mathematics (2nd Series) 28, 16–40 (1927)
12. Loos, R., Weispfenning, V.: Applying linear quantifier elimination. The Computer Journal 36, 450–462 (1993)
13. Mahboubi, A.: Contributions à la certification des calculs sur $\mathbb{R}$: théorie, preuves, programmation. PhD thesis, Université de Nice (2006)
14. McLaughlin, S., Harrison, J.: A proof-producing decision procedure for real arithmetic. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 295–314. Springer, Heidelberg (2005)
15. Nipkow, T.: Reflecting quantifier elimination for linear arithmetic. In: Grumberg, O., Nipkow, T., Pfaller, C. (eds.) Formal Logical Methods for System Security and Correctness, pp. 245–266. IOS Press, Amsterdam (2008)
16. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
17. Norrish, M.: Complete integer decision procedures as derived rules in HOL. In: Basin, D., Wolff, B. (eds.) TPHOLs 2003. LNCS, vol. 2758, pp. 71–86. Springer, Heidelberg (2003)
18. Obua, S.: Proving bounds for real linear programs in Isabelle/HOL. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 227–244. Springer, Heidelberg (2005)
19. Weispfenning, V.: The complexity of linear problems in fields. J. Symbolic Computation 5, 3–27 (1988)

Quantitative Separation Logic and Programs with Lists

Marius Bozga, Radu Iosif, and Swann Perarnau

VERIMAG, 2 Avenue de Vignate, F-38610 Gières
{iosif,bozga,perarnau}@imag.fr

Abstract. This paper presents an extension of a decidable fragment of Separation Logic for singly-linked lists, defined by Berdine, Calcagno and O'Hearn [8]. Our main extension consists in introducing atomic formulae of the form $ls^k(x,y)$ describing a list segment of length k, stretching from x to y, where k is a logical variable interpreted over positive natural numbers, that may occur further inside Presburger constraints.

We study the decidability of the full first-order logic combining unrestricted quantification of arithmetic and location variables. Although the full logic is found to be undecidable, validity of entailments between formulae with the quantifier prefix in the language $\exists^*\{\exists_{\mathbb{N}}, \forall_{\mathbb{N}}\}^*$ is decidable. We provide here a model theoretic method, based on a parametric notion of shape graphs.

We have implemented our decision technique, providing a fully automated framework for the verification of quantitative properties expressed as pre- and post-conditions on programs working on lists and integer counters.

1 Introduction

Separation Logic [15,21] has recently become a widespread formalism for the specification of programs with dynamic data structures. Due to the intrinsic complexity of the heap structures allocated and manipulated by such programs, any attempt to formalize their correctness has to be aware of the inherent bounds of undecidability. Indeed, even programs working on simple acyclic lists have the power of Turing machines, and it is expected that a general logic describing sets of configurations reached in such programs has an undecidable satisfiability (or validity) problem. An interesting problem is to define decidable logics that are either specialized for a certain kind of recursive data structures (e.g. lists, trees), or that are restricted by the quantifier prefix.

This paper presents an extension of a decidable fragment of Separation Logic for singly-linked lists, defined by Berdine, Calcagno and O'Hearn [8] and used as an internal representation for sets of states in the Smallfoot tool [4]. Our main extension consists in introducing atomic formulae of the form $ls^k(x,y)$ describing a list segment of length k, stretching from x to y, where k is a logical variable interpreted over positive natural numbers, that may occur further inside Presburger constraints. This is motivated by the need to reason about programs that work on both singly-linked list structures and integer variables (counters). We denote the extended logic as Quantitative Separation Logic (**QSL**).

In reality, many programs would traverse a list structure, while performing some iterative computation on the integer variables. The result of this computation usually depends on the number of steps, which, in turn, depends of the length of the list. A specification of the correct behavior for such a program needs to take into account both the lengths of the lists and the values of the counters.

We study the decidability properties of the full first-order logic combining unrestricted quantification of arithmetic and location variables. Although the full logic is found to be undecidable, validity of entailments between formulae with the quantifier prefix in the language $\exists^*\{\exists_{\mathbb{N}},\forall_{\mathbb{N}}\}^*$ is decidable. We provide here a model theoretic method for decidability, based on a parametric notion of shape graphs. As a byproduct, we obtain a decision procedure for the fragment of Separation Logic considered in [8].

The decision procedure for a fragment of **QSL** is currently implemented in the L2CA tool [3], a tool for translating programs with singly-linked lists into bisimilar counter automata, according to the method of [9], which opens the possibility of using well-known counter automata techniques and tools, e.g. [6,22,5], in order to verify pre- and post- conditions expressed in **QSL**, on programs working on both singly-linked lists and integer variables.

1.1 Related Work

The saga of logics for describing heap structures has its roots in the early work of Burstall [11]. Later on, work by Benedikt, Reps and Sagiv [7], Reynolds [21] and Ishtiaq and O'Hearn [15], has brought the subject into focus, whereas recent advances have been made in tackling the decidability problem [14,8,23]. The idea of combining shape and arithmetic specifications arises in the work of Zhang, Sipma and Manna [24], where a combination of free term algebras with Presburger constraints is studied for decidability. The work that is closest to ours is the one of Berdine, Calcagno and O'Hearn [8], which defines a decidable subset of Separation Logic [21] interpreted over singly-linked heap models. The work in this paper is in fact an extension of the logic in [8] with integer variables representing list lengths. One of the main challenges in the present paper was to adapt the model of parametric shape graphs in order to cope with the notion of disjunctive heaps, which is the essence of the semantic model for Separation Logic.

Regarding program analysis, the use of abstract domains (including integers and memory addresses) with quantifiers of the form $\exists^*\forall^*$ has been considered in the work of Gulwani et al. [13,12]. Unlike our approach, their work is based on using abstractions that prove to be sufficient, in general, for checking correctness of a large body of programs. Some of our examples, such as InsertSort, are also verified using the method of [12]. Recently, Magill et al. [17] report on a program analysis technique that uses Separation Logic [21] extended with first-order arithmetic. However, the main emphasis of [17] is a program analysis based on counterexample-driven abstraction refinement, whereas our work focuses on distinguishing decidable from undecidable when combining Separation Logic with first-order arithmetic. As a matter of fact, [17] claims that validity of entailments in the purely existential fragment of Separation Logic with the $ls^k(x,y)$ predicate and linear constraints is decidable, without giving the proof, by analogy to the proof-theoretic method from [8]. We extend their result by showing decidability of the validity of entailments in the $\exists^*\{\exists_{\mathbb{N}},\forall_{\mathbb{N}}\}^*$ fragment, versus undecidability of satisfiability in the $\exists^*\exists^*_{\mathbb{N}}(\forall\mid\forall_{\mathbb{N}})\exists^*\exists^*_{\mathbb{N}}$ fragment (or equivalently, validity in the $\forall^*\forall^*_{\mathbb{N}}(\exists\mid\exists_{\mathbb{N}})\forall^*\forall^*_{\mathbb{N}}$ fragment).

Roadmap The paper is organized as follows. Section 2 defines the syntax and semantics of the logic **QSL**. Section 3 proves the undecidability of the logic, while Section 4 proves the decidability of entailments in the $\exists^*\{\exists_{\mathbb{N}}, \forall_{\mathbb{N}}\}^*$ fragment. Section 5 gives some examples of programs verified using **QSL**, and Section 6 concludes. For space reasons, all proofs are given in [10].

2 Definitions

In the rest of the paper, for a set A we denote by $A_\perp$ the set $A \cup \{\perp\}$. For a function $f : A \to B$, we denote by $dom(f) = \{x \in A \mid f(x) \neq \perp\}$ its domain and by $img(f) = \{y \in B \mid \exists x \in A \,.\, f(x) = y\}$ we denote its image. The element $\perp$ is used to denote that a (partial) function is undefined at a given point, e.g. $f(x) = \perp$. Sometimes we shall use the graph notation for functions, i.e. $f = \{\langle a,b \rangle, \ldots\}$ if $f(a) = b, \ldots$, etc. The notation $\lambda x : A.y$ stands for the function $\{\langle x,y \rangle \mid x \in A\}$, and $\lambda x : A.\perp$ is the empty function $\emptyset$, by convention. Let $Part(S)$ denote the set of all partitions of the set S.

By $\mathcal{T}(X)$ we denote the set of all terms build using variables $x \in X$. For a term (formula) $\tau(X)$ and a mapping $\mu : X \to \mathcal{T}(X)$, we denote by $\tau[\mu]$ the term (formula) in which each occurrence of x is replaced with $\mu(x)$. For a formula φ, we denote as $FV(\varphi)$ the set of its free variables. If φ is a formula of the first-order arithmetic of integers, and $\nu : FV(\varphi) \to \mathbb{Z}$ is an interpretation of its free variables, we denote by $\nu \models \varphi$ the fact that $\varphi[\nu]$ is a valid formula.

Presburger arithmetic $\langle \mathbb{N}, +, 0, 1 \rangle$ is the theory of first-order logic of addition and successor function ($S(n) = n+1$) [20]. The interpretation of logical variables is the set of natural numbers $\mathbb{N}$, and the meaning of the function symbols $0, 1, +$ is the natural one. It is well-known that the satisfiability problem for Presburger arithmetic is decidable [20].

$u, v, \ldots \in$	$PVar$	program variables
$x, y, \ldots \in$	$LVar$	location variables
$k, l, \ldots \in$	$IVar$	integer variables
L	$:= \mathrm{nil} \mid u \mid x$	location expressions
I	$:= n \in \mathbb{N} \mid k \mid I + I$	integer expressions
A	$:= I = I \mid L = L \mid \mathrm{emp} \mid L \mapsto L \mid ls^I(L,L)$	atomic propositions
F	$:= \mathbb{T} \mid A \mid \neg F \mid F \wedge F \mid F * F \mid \exists x \,.\, F \mid \exists_{\mathbb{N}} k \,.\, F$	formulae

Fig. 1. Separation Logic with Presburger Arithmetic

The syntax of **QSL** is given in Figure 1. Notice the difference between program variables $PVar$ and location variables $LVar$, the former being logical constants, whereas the latter may occur within the scope of a quantifier. Let $FV_L(\varphi) = FV(\varphi) \cap LVar$ and $FV_I(\varphi) = FV(\varphi) \cap IVar$ denote the sets of location and integer free variables of φ, respectively.

As usual, we define $\varphi \vee \psi \stackrel{\Delta}{=} \neg(\neg\varphi \wedge \neg\psi)$, $\varphi \Rightarrow \psi \stackrel{\Delta}{=} \neg\varphi \vee \psi$, $\forall x \,.\, \varphi \stackrel{\Delta}{=} \neg\exists x \,.\, \neg\varphi$ and $\forall_{\mathbb{N}} k \,.\, \varphi \stackrel{\Delta}{=} \neg\exists_{\mathbb{N}} k \,.\, \neg\varphi$. Moreover, we write $k \leq l$ and $ls(x,y)$ as shorthands for $\exists_{\mathbb{N}} k' \,.\, k + k' = l$ and $\exists_{\mathbb{N}} k \,.\, ls^k(x,y)$, respectively. $\mathbb{F}$ is a shorthand for $\neg\mathbb{T}$. The bounded quantifiers $\exists_{\mathbb{N}} m \leq n \,.\, \varphi(m)$ and $\forall_{\mathbb{N}} m \leq n \,.\, \varphi(m)$ are used instead of $\exists_{\mathbb{N}} m \,.\, m \leq n \wedge \varphi(m)$ and $\forall_{\mathbb{N}} m \,.\, m \leq n \Rightarrow \varphi(m)$, respectively. We shall also deploy some of the classical shorthands in Separation Logic: $x \mapsto \_ \stackrel{\Delta}{=} \exists y \,.\, x \mapsto y$, and $x \hookrightarrow y \stackrel{\Delta}{=} x \mapsto y * \mathbb{T}$, where y is either a location variable or nil. For list segment formulae we define $\widetilde{ls}^k(x,y) \stackrel{\Delta}{=} ls^k(x,y) * \mathbb{T}$ and $\widetilde{ls}(x,y) \stackrel{\Delta}{=} ls(x,y) * \mathbb{T}$.

The semantics of **QSL** formulae is given in terms of heaps. A *heap* is a rooted graph in which each node has at most one successor. Let *Loc* denote the set of *locations*. We assume henceforth that *Loc* is an infinite, countable set, with a designated element $nil \in Loc$. In what follows, we identify heaps that differ only by a renaming of their locations.

Definition 1. *A* heap *is a pair $H = \langle s,h \rangle$, where $s : PVar \cup LVar \to Loc_\perp$ associates variables with locations, and $h : Loc \to Loc_\perp$ is the partial successor mapping. In particular, we have $h(nil) = \perp$. We denote by $\mathcal{H}$ the set of all heaps with variables from $PVar \cup LVar$ and locations from Loc.*

The interpretation of a formula is defined by a forcing relation $\models$ between tuples $\langle H, \nu, \iota \rangle \in \mathcal{H} \times (LVar \mapsto Loc_\perp) \times (IVar \mapsto \mathbb{N}_\perp)$ and formulae. Here $\nu : LVar \to Loc_\perp$ is a partial valuation of location variables, and $\iota : IVar \to \mathbb{N}_\perp$ is a partial valuation of integer variables. The semantics of **QSL** formulae is given below, for a given heap $H = \langle s,h \rangle$:

$$[\![u]\!]_{\langle H,\nu \rangle} = s(u),\ [\![x]\!]_{\langle H,\nu \rangle} = \nu(x),\ [\![\mathrm{nil}]\!]_{\langle H,\nu \rangle} = nil$$

$\langle H,\nu,\iota \rangle \models \mathbb{T}$		always
$\langle H,\nu,\iota \rangle \models L_1 = L_2$	iff	$[\![L_1]\!]_{\langle H,\nu \rangle} = [\![L_2]\!]_{\langle H,\nu \rangle}$
$\langle H,\nu,\iota \rangle \models \mathrm{emp}$	iff	$h = \emptyset$
$\langle H,\nu,\iota \rangle \models L_1 \mapsto L_2$	iff	$h = \{\langle [\![L_1]\!]_{\langle H,\nu \rangle}, [\![L_2]\!]_{\langle H,\nu \rangle} \rangle\}$
$\langle H,\nu,\iota \rangle \models \neg\varphi$	iff	$\langle H,\nu,\iota \rangle \not\models \varphi$
$\langle H,\nu,\iota \rangle \models \varphi \wedge \psi$	iff	$\langle H,\nu,\iota \rangle \models \varphi$ and $\langle H,\nu,\iota \rangle \models \psi$
$\langle H,\nu,\iota \rangle \models \varphi * \psi$	iff	there exist H_1, H_2 such that $H = H_1 \bullet H_2$ and $\langle H_1,\nu,\iota \rangle \models \varphi, \langle H_2,\nu,\iota \rangle \models \psi$
$\langle H,\nu,\iota \rangle \models \exists x \,.\, \varphi$	iff	$\langle H,\nu[x \leftarrow l],\iota \rangle \models \varphi$ for some $l \in Loc \setminus \{nil\}$

Here $H_1 \bullet H_2$ denotes the disjoint union of $H_1 = \langle s,h_1 \rangle$ and $H_2 = \langle s,h_2 \rangle$, i.e, $dom(h_1) \cap dom(h_2) = \emptyset$, $h = h_1 \cup h_2$. The above definitions are standard in Separation Logic [21]. The rules below are specific to our extension:

$$[\![I]\!]_\iota = I[\iota]$$

$\langle H,\nu,\iota \rangle \models I_1 = I_2$	iff	$[\![I_1]\!]_\iota = [\![I_2]\!]_\iota$
$\langle H,\nu,\iota \rangle \models ls^0(L_1,L_2)$	iff	$\langle H,\nu,\iota \rangle \models L_1 = L_2 \wedge \mathrm{emp}$
$\langle H,\nu,\iota \rangle \models ls^{n+1}(L_1,L_2)$	iff	$\langle H,\nu,\iota \rangle \models \exists x \,.\, ls^n(L_1,x) * x \mapsto L_2$
$\langle H,\nu,\iota \rangle \models ls^I(x,y)$	iff	$\langle H,\nu,\iota \rangle \models ls^{[\![I]\!]_\iota}(x,y)$
$\langle H,\nu,\iota \rangle \models \exists_{\mathbb{N}} k \,.\, \varphi$	iff	$\langle H,\nu,\iota[k \leftarrow n] \rangle \models \varphi$, for some $n \in \mathbb{N}$

There are two types of quantifiers, $\exists$ ranges over locations Loc, and $\exists_{\mathbb{N}}$ over natural numbers $\mathbb{N}$. A tuple $\langle H, \nu, \iota \rangle$ is said to be a *model* of φ iff $\langle H, \nu, \iota \rangle \models \varphi$. If $FV(\varphi) = \emptyset$, we denote the fact that H is a model of φ directly as $H \models \varphi$. An *entailment* is a formula of type $\varphi \Rightarrow \psi$. Given such an entailment, the *validity problem* asks if it holds for any tuple $\langle H, \nu, \iota \rangle$, i.e. if any model of φ is also a model of ψ.

Note that we use the "classical" (non-intuitionistic) semantics of Separation Logic [15], in which a points-to relation $L_1 \mapsto L_2$ is true iff the heap is defined on only one cell whose address is the value of L_1. As a result, $ls^k(L_1, L_2)$ is true iff the heap is defined only on the set of addresses that form the list from L_1 up to (but not including) L_2. Another consequence of using this semantics is that $ls^0(L_1, L_2)$ is true only on the empty heap. One can however recover the intuitionistic semantics of [21] by using the shorthands $L_1 \hookrightarrow L_2$ and $\widetilde{ls}^k(L_1, L_2)$ instead.

The following notion of *dangling location* is essential for the semantics of Separation Logic on heaps [15,21]. To understand this point, consider the formula $\varphi : (u \mapsto v) * (v \mapsto nil)$, describing a heap $H = \langle s, h \rangle$, in which u and v are allocated to two different cells, i.e. $s(u) = l_1$, $s(v) = l_2$, and nothing else is in the domain of the heap, i.e. $h = \{\langle l_1, l_2 \rangle, \langle l_2, nil \rangle\}$. The reason for which $H \models \varphi$, is that there exists two disjoint heaps, namely $H_1 = \langle s, \{\langle l_1, l_2 \rangle\} \rangle$ and $H_2 = \langle s, \{\langle l_2, nil \rangle\} \rangle$, such that $H_1 \models u \mapsto v$ and $H_2 \models v \mapsto nil$. Notice the role of the location l_2, pointed to by the variable v, which is referenced by the first heap, but allocated in the second one. This location ensures that the disjoint union of H_1 and H_2 is defined, and that $H_1 \bullet H_2 \models u \mapsto v * v \mapsto \text{nil}$.

Definition 2. *A location $l \in Loc \setminus \{nil\}$ is said to be* dangling *in a heap $H = \langle s, h \rangle$ iff $l \in (img(s) \cup img(h)) \setminus dom(h)$.*

In the following, we denote by $dng(H)$ the set of all dangling nodes of H, and by $loc(H) = img(s) \cup dom(h) \cup img(h)$ the set of all locations, either defined or dangling in H.

2.1 Motivating Example

Let us consider the program in Figure 2. The loop on the left hand side inserts elements into the list pointed to by u, while incrementing the c counter, and the loop on the right removes the elements in reversed order, while decrementing c. The pre- and post-condition of the program are inserted as Hoare-style annotations. Both initially and finally, the value of c is zero and the heap is empty.

```
{$c = 0 \wedge \mathbf{emp} \wedge u = \text{nil}$}
1: while ... do                              7: while $c \neq 0$ do
{$c \geq 0 \wedge c = k \wedge \mathbf{ls}^k(u, nil)$}    {$c > 0 \wedge c = k \wedge \mathbf{ls}^k(u, nil)$}
2:      t := new;                            8:      u := u.next;
3:      t.next := u;                         9:      c := c - 1;
4:      u := t;                              10: od
5:      c := c + 1;                          {$c = 0 \wedge \mathbf{emp}$}
6: od
```

Fig. 2. Program verification using **QSL**

In order to prove that the program terminates without a null pointer dereferencing, and moreover ensuring that the post-condition holds, one needs to relate the value of c to the length of the list pointed to by u, as it is done in the invariants of the left and right hand side : $c = k \wedge \mathbf{ls}^k(u, nil)$. This example could not be handled using standard Separation Logic, since we explicitly need the ability of reasoning about both list lengths and integer variables.

3 Undecidability of QSL

In this section we prove the undecidability of the **QSL** logic. Namely the class of formulae with quantifier prefix in the language $\exists^*\exists^*_{\mathbb{N}}(\forall \mid \forall_{\mathbb{N}})\exists^*\exists^*_{\mathbb{N}}$ are shown to have an undecidable satisfiability problem. It is to be noticed that undecidability of **QSL** is not a direct consequence of the undecidability of Separation Logic [19], since the proof in [19] uses multiple selector heaps, while in this case we consider only heaps composed of singly-linked lists. Our result is non-trivial since it is well-known also that, even simple logics, e.g. FOL, MSOL are decidable when interpreted over singly-linked lists, and become quickly undecidable when interpreted over grid-like, and more general graph structures.

Theorem 1. *The set of* **QSL** *formulae which, written in prenex normal form, have the quantifier prefix in the language* $\exists^*\exists^*_{\mathbb{N}}(\forall \mid \forall_{\mathbb{N}})\exists^*\exists^*_{\mathbb{N}}$*, is undecidable.*

The idea of the proof is that one can encode all terminating runs of an arbitrary 2-counter machine [18] by a formula of **QSL** in the $\exists^*\exists^*_{\mathbb{N}}(\forall \mid \forall_{\mathbb{N}})\exists^*\exists^*_{\mathbb{N}}$ quantifier fragment. Since the halting problem is undecidable for 2-counter machines, the satisfiability of formulae in the above mentioned fragment is also undecidable.

In the following developments, we shall prove that logical entailment in the $\exists^*\{\exists_{\mathbb{N}}, \forall_{\mathbb{N}}\}^*$ fragment of **QSL** is decidable. We have found no argument for (un)decidability concerning the quantifier prefix fragment $\{\exists, \forall\}^*\{\exists_{\mathbb{N}}, \forall_{\mathbb{N}}\}^*$. In particular, all attempts to reduce (from) to known fragments of MSO with cardinality constraints [16] have failed.

4 Model Theoretic Method

The validity of an entailment $\varphi \Rightarrow \psi$ is equivalent to the non-satisfiability of the formula $\varphi \wedge \neg\psi$, i.e. there should be no tuples $\langle H, \nu, \iota \rangle$ such that $\langle H, \nu, \iota \rangle \models \varphi$ and $\langle H, \nu, \iota \rangle \not\models \psi$. Our main result, leading immediately to decidability of entailments, is that, if φ is of the form $\exists x_1 \ldots \exists x_n Q_1 l_1 \ldots Q_m l_m \;.\; \theta(\mathbf{x}, \mathbf{l})$, with $Q_i \in \{\exists_{\mathbb{N}}, \forall_{\mathbb{N}}\}$ and θ is a boolean combination of predicates with $\neg$, $\wedge$ and $*$, all models $\langle H, \nu, \iota \rangle$ of φ can be represented using a finite number of (finite) structures called symbolic graph representations (SGR).

The decision procedure for the validity of a QSL entailment $\varphi \Rightarrow \psi$ is based on the following idea. We first define operators on sets of SGRs that are the counterparts of the logical connectives $\vee$, $\wedge$, $*$ and the existential quantifiers $\exists x$, $\exists_{\mathbb{N}} x$. Second, for each existential QSL formula φ, we compute a set $[\![\varphi]\!]$ of SGRs that represent all models of φ. The construction of this set is recursive, on the structure of φ. The entailment $\varphi \Rightarrow \psi$

is valid iff the set $[\![\varphi]\!] \ominus [\![\psi]\!]$ is empty, where $\ominus$ is an operator defined on SGRs, that computes the representation of the difference between the set of concrete models of the φ and the one of ψ. Least, the emptiness problem for sets of SGRs is shown to be decidable, by reduction to the satisfiability problem for the Presburger arithmetic.

4.1 Symbolic Shape Graphs

In this section we define a finite representation of (possibly infinite) sets of heaps, called symbolic shape graphs (SSG), which is the essence of our decision method. The next section defines the SGR representation for sets of heaps, which is based on SSGs and arithmetic constraints.

Definition 3. *Given a heap $H = \langle s, h\rangle \in \mathcal{H}$, a location $l \in Loc$ is said to be a* cut point *in H if either $l \in img(s) \cup dng(H) \cup \{nil\}$, or there exists two distinct locations $l_1, l_2 \in Loc$ such that $h(l_1) = h(l_2) = l$.*

A location l is a cut point in a heap if either (1) l is pointed to directly by a program variable, i.e. $l \in img(s)$, (2) l is dangling or *nil*, or (3) l has more than one predecessor in the heap. We denote by $l_1 \rhd_H l_2$ the fact that $h(l_1) = l_2$ and $l_2 \neq \perp$ is not a cut point in H. Let $\sim_H$ denote the reflexive, symmetric and transitive closure of the $\rhd_H$ relation, i.e. the smallest equivalence relation that includes $\rhd_H$, and $[l]_\sim$ be the equivalence class of $l \in Loc$ w.r.t. $\sim_H$. We also refer to these equivalence classes as to *list segments*. By convention, we have $[\perp]_\sim = \perp$. Let $H_{/\sim} = \langle s_{/\sim}, h_{/\sim}\rangle$ be the *quotient heap*, where:

- $s_{/\sim} : PVar \cup LVar \rightarrow Loc_{/\sim_\perp}$ and $s_{/\sim}(u) = [s(u)]_\sim$, for all $u \in PVar$,
- $h_{/\sim} : Loc_{/\sim} \rightarrow Loc_{/\sim_\perp}$ and for all $l \in dom(h)$, if $h(l) = l'$ and l' is either $\perp$ or a cut point in $\langle s, h\rangle$, then $h_{/\sim}([l]) = [l']_\sim$. In particular, $h_{/\sim}([l]) = \perp$, for all $l \notin dom(h)$.

Note that $s_{/\sim}$ and $h_{/\sim}$ are well-defined functions. We extend the rest of notations to quotient heaps, i.e. $dng(H_{/\sim}) = \{[l]_\sim \mid l \in dng(H)\}$ and $loc(H/\sim) = \{[l]_\sim \mid l \in loc(H)\}$.

For example, in the heap from Figure 3 (a), the cut points are marked by hollow nodes and the $\sim$-equivalence classes are enclosed in solid boxes. The quotient heap is the heap in which these boxes are taken as nodes, instead of the individual locations.

Definition 4. *Given a set PVar of program variables, a set LVar of location variables, and a set of* counters $\mathcal{Z} = \{z_1, \ldots, z_n\}$, *a* symbolic shape graph *(SSG) is a tuple $G = \langle N, D, R, Z, S, V\rangle$, where:*

- *N is a finite set of symbolic nodes, with a designated node $Nil \in N$,*
- *$D \subseteq N$ is a set of symbolic dangling nodes,*
- *$R \subseteq N$ is a set of symbolic root nodes,*
- *$Z : N \setminus D \rightarrow \mathcal{Z}$ is an injective function assigning each non-dangling node to a counter,*
- *$S : N \rightarrow N_\perp$ is the successor function, where:*
  - *$S(Nil) = \perp$ and $S(d) = \perp$, for all $d \in D$,*
  - *$S(n) \notin R$, for all $n \in N$,*
  - *$S(n) \in N$, for all $n \in N \setminus (D \cup \{Nil\})$.*
- *$V : PVar \cup LVar \rightarrow N$ assigns program and location variables with nodes.*

Intuitively, each node of a SSG represents a list segment of a concrete heap. The node *Nil* stands for the concrete *nil* location, and each symbolic dangling node represents one dangling location.

Definition 5. *An SSG $G = \langle N, D, R, Z, S, V\rangle$ is said to be in* normal form *if:*

- *each node in $n \in N \setminus \{Nil\}$ is reachable either from $V(u)$, for some $u \in PVar \cup LVar$, or from some symbolic root $r \in R$, and*
- *either $n \in img(V) \cup D \cup \{Nil\}$, or there exist two distinct nodes $n_1, n_2 \in N$ such that $S(n_1) = S(n_2) = n$.*

$\mathcal{S}_k$ denotes the set of SSGs in normal form, with $|R| \leq k$ and $img(V) \subseteq PVar \cup LVar$.

Sometimes we denote by $\mathcal{S}$ the union $\bigcup_{k \in \mathbb{N}} \mathcal{S}_k$. We identify SSGs which are equivalent under renaming of nodes and counters. The following was proved in [9]:

Lemma 1. *Let $G = \langle N, D, R, Z, S, V\rangle \in \mathcal{S}_k$ be a normal form SSG. Then, $|N| \leq 2(|dom(V)| + |R|)$. As a consequence, the number of such SSGs is bounded asymptotically by $2(|PVar| + |LVar| + k)^{2(|PVar|+|LVar|+k)}$, and the bound is tight.*

The following definition relates the notions of heap and SSG.

Definition 6. *Let $G = \langle N, D, R, Z, S, V\rangle \in \mathcal{S}$ be a SSG, $\nu : dom(V) \cap LVar \rightarrow dom(h)$ a valuation of the location variables of G, and $\iota : img(Z) \rightarrow \mathbb{N}^+$ a valuation of the counters in G. Let $H = \langle s, h\rangle \in \mathcal{H}$ be a heap such that $dom(s) = dom(V) \cap PVar$, and $H_{/\sim} = \langle s_{/\sim}, h_{/\sim}\rangle$ be the quotient of H with respect to $\sim_H$. We say that H is* the $\langle \nu, \iota\rangle$-concretization *of G iff there exists a bijective mapping $\eta : N_\perp \rightarrow (loc(h_{/\sim}) \cup \{nil, \perp\})$ such that:*

- *$\eta(Nil) = \{nil\}$ and $\eta(\perp) = \perp$,*
- *$\eta(V(u)) = s_{/\sim}(u)$, for all $u \in PVar$,*
- *$\eta(S(n)) = h_{/\sim}(\eta(n))$, for all $n \in N \setminus D$,*
- *$\eta(n) \in dng(H_{/\sim})$, for all $n \in D$,*
- *$\iota(Z(n)) = |\eta(n)|$, for all $n \in N \setminus D$.*

We recall upon the fact that heaps are identical, up to isomorphism, which implies that a $\langle \nu, \iota\rangle$ is uniquely defined. We say that H is a concretization of G if there exist ν, ι such that H is the $\langle \nu, \iota\rangle$-concretization of G. Roughly speaking, the $\langle \nu, \iota\rangle$-concretization of a SSG G is the heap obtained by replacing each node n of G with a list segment whose length equals the value of the counter $Z(n)$. Moreover, if G has a $\langle \nu, \iota\rangle$-concretization, we must have $\iota(Z(n)) > 0$, for all non-dangling symbolic nodes $n \in N$. Notice also that dangling locations are represented by symbolic dangling nodes. We denote by $\gamma_{\nu,\iota}(G)$ the $\langle \nu, \iota\rangle$-concretization of G and by $\Gamma(G)$ the set of all concretizations of G.

For example, the SSG in Figure 3 (b) has as $\langle \nu, \iota\rangle$-concretization the heap in Figure 3 (a), for the valuations:

- $\nu(x) = l_3$, $\nu(y) = l_{10}$, and
- $\iota(z_1) = 2$, $\iota(z_2) = 1$, $\iota(z_3) = 3$, $\iota(z_4) = 2$, $\iota(z_5) = 1$, $\iota(z_6) = 3$.

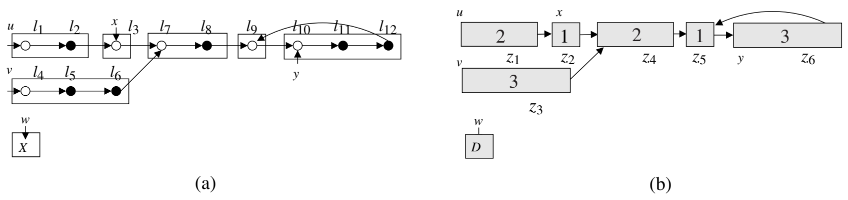


Fig. 3. SSG and Concretization

Notice that the symbolic dangling node pointed to by w corresponds to a dangling location pointed to by w in Figure 3 (a).

The following result expresses the fact that one heap may not be the concretization of two different (non-isomorphic) SSGs:

Lemma 2. *For two non-isomorphic SSGs $G_1, G_2 \in \mathcal{S}$, we have $\Gamma(G_1) \cap \Gamma(G_2) = \emptyset$.*

4.2 Symbolic Graph Representations

In this section we introduce the notion of symbolic graph representation (SGR) together with a number of operators on these structures. In the next section, we shall provide a stepwise translation of a **QSL** formula with quantifier prefix $\exists^*\{\exists_{\mathbb{N}}, \forall_{\mathbb{N}}\}^*$ into a set of symbolic graph representations.

A *symbolic graph representation* is a pair $\langle G, \varphi\rangle$, where $G = \langle N, D, R, Z, S, V\rangle$ is a SSG in normal form and φ an open formula over the counters of G, i.e. $FV(\varphi) \subseteq img(Z)$. By $\mathcal{G}$ we denote the set of all SGRs $R = \langle G, \varphi\rangle$, where $G \in \mathcal{S}$ and the set of counters in each G is a subset of $\mathcal{Z}$.

A heap $H = \langle s, h\rangle$ is the $\langle \nu, \iota\rangle$-concretization of $\langle G, \varphi\rangle$ iff $\nu : dom(V) \cap LVar \to dom(h)$ is a valuation of the location variables of G, and $\iota : img(Z) \to \mathbb{N}^+$ is a valuation of the counters in G that satisfies φ, i.e. $\iota \models \varphi$. This is denoted in the following as $H = \gamma_{\nu,\iota}(\langle G, \varphi\rangle)$. $\Gamma(\langle G, \varphi\rangle)$ denotes the set of all $\langle \nu, \iota\rangle$-concretizations of $\langle G, \varphi\rangle$. The notation is lifted to finite sets of SGRs in the obvious way: $\Gamma(\{R_1, \ldots, R_n\}) = \bigcup_{i=1}^{n} \Gamma(R_i)$.

We introduce now three operators on finite sets of SGRs, that correspond to the boolean operators of union, intersection and set difference. Let $S_1, S_2 \subseteq \mathcal{G}$ be two finite sets of SGRs.

$$\begin{aligned} S_1 \sqcup S_2 = \quad & \{\langle G, \varphi_1 \vee \varphi_2\rangle \mid \langle G, \varphi_1\rangle \in S_1 \text{ and } \langle G, \varphi_2\rangle \in S_2\} \cup \\ & \{\langle G, \varphi\rangle \in S_1 \mid \langle G, \_\rangle \notin S_2\} \cup \{\langle G, \varphi\rangle \in S_2 \mid \langle G, \_\rangle \notin S_1\} \\ S_1 \sqcap S_2 = \quad & \{\langle G, \varphi_1 \wedge \varphi_2\rangle \mid \langle G, \varphi_1\rangle \in S_1 \text{ and } \langle G, \varphi_2\rangle \in S_2\} \\ S_1 \ominus S_2 = \quad & \{\langle G, \varphi_1 \wedge \neg\varphi_2\rangle \mid \langle G, \varphi_1\rangle \in S_1 \text{ and } \langle G, \varphi_2\rangle \in S_2\} \\ & \cup \{\langle G, \varphi\rangle \in S_1 \mid \langle G, \_\rangle \notin S_2\} \end{aligned}$$

Here the notation $\langle G, \_\rangle$ stands for any SGR pair having G as its first component. Let $G = \langle N, D, R, Z, S, V\rangle$ and notice that, since $FV(\varphi_1) \subseteq img(Z)$ and $FV(\varphi_2) \subseteq img(Z)$, then $FV(\varphi_1 \vee \varphi_2)$, $FV(\varphi_1 \wedge \varphi_2)$ and $FV(\varphi_1 \wedge \neg\varphi_2)$ are also subsets of $img(Z)$.

Lemma 3. *SGRs are effectively closed under union, intersection and difference. In particular, we have* $\Gamma(S_1 \sqcup S_2) = \Gamma(S_1) \cup \Gamma(S_2)$, $\Gamma(S_1 \sqcap S_2) = \Gamma(S_1) \cap \Gamma(S_2)$ *and* $\Gamma(S_1 \ominus S_2) = \Gamma(S_1) \setminus \Gamma(S_2)$.

The $\circledast$ operator is defined on SGRs with the following meaning : for two SGRs R_1 and R_2, we have $\Gamma(R_1 \circledast R_2) = \{H_1 \bullet H_2 \mid H_1 \in \Gamma(R_1) \text{ and } H_2 \in \Gamma(R_2)\}$. In other words, $\circledast$ is the SGR counterpart of the disjoint union operator on heaps. However, $\circledast$ is not a total operator, i.e. it is not defined for any pair of SGRs, but only for the ones complying with the following definition :

Definition 7. *Two SSGs* $G_i = \langle N_i, D_i, Z_i, S_i, V_i \rangle$, $i = 1,2$ *are said to* match *iff there exists a mapping* $\mu : D_1 \cup D_2 \to (N_1 \cup N_2)_\perp$ *such that, for all* $u \in dom(V_1) \cap dom(V_2)$*, either:*

- $V_1(u) \in D_1$ *and* $\mu(V_1(u)) = V_2(u)$*, or*
- $V_2(u) \in D_2$ *and* $\mu(V_2(u)) = V_1(u)$.

and $\mu(d) = \perp$*, for all* $d \in (D_1 \cup D_2) \setminus (dom(V_1) \cap dom(V_2))$.

Intuitively, two SSGs match if it is possible to relate any dangling node pointed to by program variable in one SSG to a node pointed to by the same variable in the other SSG. Note that two SSGs do not match if the same variable points to some non-dangling node in both. Figure 4 gives an example of two matching SSGs (a) and (b) together with the mapping μ between their nodes (in dotted lines). According to Definition (7), the choice of μ is not unique.

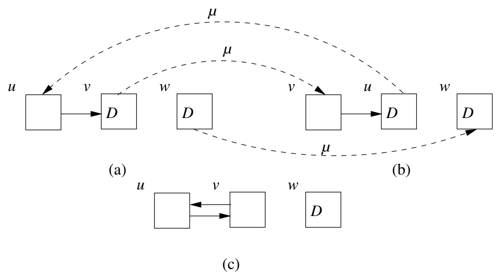


Fig. 4. Matching SSGs

Given two SGRs $R_1 = \langle G_1, \varphi_1 \rangle$ and $R_2 = \langle G_2, \varphi_2 \rangle$, with *matching* underlying SSGs $G_i = \langle N_i, D_i, Z_i, S_i, V_i \rangle$, (for the purposes of this definition, we can assume w.l.o.g. that $N_1 \cap N_2 = \{Nil\}$ and $img(Z_1) \cap img(Z_2) = \emptyset$), we define $R_1 \circledast R_2 = \langle G, \varphi_1 \wedge \varphi_2 \rangle$, $G = \langle N, D, Z, S, V \rangle$, where:

- $N = (N_1 \cup N_2) \setminus dom(\mu)$,
- $D = (D_1 \cup D_2) \setminus dom(\mu)$,
- $R = (R_1 \cup R_2) \setminus dom(\mu)$,
- $Z = Z_1 \cup Z_2$,

– for all $n \in N$:

$$S(n) = \begin{cases} S_i(n) & \text{if } n \in N_i \text{ and } S_i(n) \notin dom(\mu) \\ \mu(S_i(n)) & \text{if } n \in N_i \text{ and } S_i(n) \in dom(\mu) \end{cases} \quad i = 1,2$$

– for all $u \in dom(V_1) \cup dom(V_2)$:

$$V(u) = \begin{cases} V_i(u) & \text{if } V_i(u) \notin dom(\mu) \\ \mu(V_i(u)) & \text{if } V_i(u) \in dom(\mu) \end{cases} \quad i = 1,2$$

For example, the SSG in Figure 4 (c) is the result of the $\circledast$-composition of the SSGs in Figure 4 (a) and (b).

The $\circledast$ operator is undefined, if G_1 and G_2 do not match. Notice that if $G_1 \in \mathcal{S}_{k_1}$, $G_2 \in \mathcal{S}_{k_2}$ and $\langle G, \varphi \rangle = \langle G_1, \varphi_1 \rangle \circledast \langle G_2, \varphi_2 \rangle$, then $G \in \mathcal{S}_{k_1+k_2}$. The correctness of the definition is captured by the following Lemma:

Lemma 4. *Given two SGRs $R_1 = \langle G_1, \varphi_1 \rangle$ and $R_2 = \langle G_2, \varphi_2 \rangle$, such that G_1 and G_2 match, we have $\Gamma(R_1 \circledast R_2) = \{H_1 \bullet H_2 \mid H_1 \in \Gamma(R_1),\ H_2 \in \Gamma(R_2)\}$.*

The following *projection* operator captures the effect of dropping one location variable out of the heap. Let $R = \langle G, \varphi \rangle$ be an SGR, where $G = \langle N, D, R, Z, S, V \rangle$ is the underlying SSG, and $x \in img(V) \cap LVar$ be a location variable occurring in G. For an arbitrary symbolic node $n \in N$, let $prec_G(n) = \{m \in N \mid m \neq n,\ S(m) = n\}$ be the set of predecessors of n, different from itself, in G.

We define $R\downarrow_x$ to be the SGR, having a normal-form underlying SSG (cf. Definition 4), from which x is missing. Formally, let $R\downarrow_x = \langle G', \varphi' \rangle$, where:

1. if $x \notin dom(V)$ then $G' = G$ and $\varphi' = \varphi$.
2. else, if $x \in dom(V)$ and either:
   (a) there exists $u \in dom(V) \setminus \{x\}$ such that $V(u) = V(x)$, or
   (b) there exist $m_1, m_2 \in dom(S)$ s.t. $m_1 \neq m_2$ and $S(m_1) = S(m_2) = V(x)$
   then $G' = \langle N, D, R, Z, S, V[x \leftarrow \bot] \rangle$ and $\varphi' = \varphi$.
3. else, if $x \in dom(V)$, $V(x) = n$, and for all $u \in dom(V) \setminus \{x\}$, we have $V(u) \neq n$, and either:
   (a) $prec_G(n) = \emptyset$, then $G' = \langle N, D, R \cup \{n\}, Z, S, V[x \leftarrow \bot] \rangle$ and $\varphi' = \varphi$, or
   (b) $n \in D$ and $prec_G(n) \neq \emptyset$, then $G' = \langle N, D, R, Z, S, V[x \leftarrow \bot] \rangle$ and $\varphi' = \varphi$,
   (c) $n \notin D$ and $m \in prec_G(n)$, where $Z(m) = k_1$ and $Z(n) = k_2$, then
   $G' = \langle N \setminus \{n\}, D, R, Z[m \leftarrow k_3][n \leftarrow \bot], S[m \leftarrow S(n)][n \leftarrow \bot], V[x \leftarrow \bot] \rangle$ and
   $\varphi' = \exists k_1 \exists k_2 \,.\, \varphi \wedge k_3 = k_1 + k_2$, where $k_3 \notin img(Z)$ is a fresh counter name.

The correctness of this definition is captured in the following Lemma:

Lemma 5. *Let $R = \langle G, \varphi \rangle$ be a SGR, $G = \langle N, D, Z, S, V \rangle$ be its underlying SSG, and $x \in LVar$ be a location variable. Then $\Gamma(R\downarrow_x) = \{\langle s[x \leftarrow \bot], h \rangle \mid \langle s, h \rangle \in \Gamma(R)\}$.*

Given a set S of SGRs, the emptiness problem $\Gamma(S) = \emptyset$ is effectively decidable if all constraints φ occurring within elements $\langle G, \varphi \rangle \in \mathcal{G}$ are written in a logic decidable for satisfiability. In our case, this logic is the Presburger arithmetic, for which the satisfiability problem is known to be decidable [20].

4.3 From Formulae to Sets of SGR

We are now ready to describe the construction of a set of SGRs for a given formula :

$$\varphi \ : \ \exists x_1 \ldots \exists x_n Q_1 l_1 \ldots Q_m l_m \ . \ \theta(\mathbf{x},\mathbf{l})$$

where $Q_i \in \{\exists_{\mathbb{N}}, \forall_{\mathbb{N}}\}$ and θ is a quantifier-free QSL formula. The construction is performed incrementally, following the structure of the abstract syntax tree of θ. The set $\mathbf{x} = \{x_1,\ldots,x_n\}$ is called from now on the *support set* of θ. Without losing generality, we consider that the leaves of this tree are atomic propositions of one of the forms : $\mathbb{T}$, emp, $x = y$, $x \mapsto y$ and $ls^l(x,y)$, where $x \in \mathbf{x} \cup PVar$, $y \in \mathbf{x} \cup PVar \cup \{nil\}$ and $l \in \{l_1,\ldots,l_m\}$.

From now on, let $\mathcal{S}_k(\mathbf{x})$ be the set of all SSGs with at most k root nodes, support variables from $PVar \cup \mathbf{x}$, and counters from a fixed given set $\mathcal{Z}$. Given a formula φ, we denote by $[\![\varphi]\!]_{\mathbf{x}}(k)$ the set of SGRs with at most k root nodes, over the support set $\mathbf{x}$, defining the models of φ, in the following sense. The set of concrete heaps corresponding to $[\![\varphi]\!]_{\mathbf{x}}(k)$ is exactly the set of models of φ.

For atomic spatial propositions, $[\![\varphi]\!]_{\mathbf{x}}(k)$ is computed according to the definitions from Table 1. In the definition of $[\![\text{emp}]\!]_{\mathbf{x}}(k)$ we consider as parameter the partition $\langle Y_1,\ldots,Y_p\rangle \in Part(\mathbf{x})$. That is $[\![\text{emp}]\!]_{\mathbf{x}}(k) = \bigcup_{\langle Y_1,\ldots,Y_p\rangle \in Part(\mathbf{x})} [\![\text{emp}]\!]^{Y_1,\ldots,Y_p}$, where $[\![\text{emp}]\!]^{Y_1,\ldots,Y_p}$ is defined in Table 1. Intuitively, Y_i, $1 \leq i \leq p$ is the set of variables that are aliased, pointing to the same dangling node d_i, in the empty heap. In Table 1, let $D = \{d_1,\ldots,d_{p-1}\}$, $R = \emptyset$ and $\Delta = \bigcup_{i=1}^{p-1} \lambda x : Y_i.d_i \cup \lambda x : Y_p.Nil$.

Since emp denotes all heaps with empty domain, in the SGR representation there are no symbolic nodes, which are not dangling or nil. Moreover, there are no counters, and therefore, no arithmetic constraints.

In the definition of $[\![\varphi]\!]_{\mathbf{x}}(k)$ for $x \mapsto nil$ and $x \mapsto y$, we consider two parameters: (1) a set $Z \subseteq \mathbf{x} \cup \{x\}$, such that $x \in Z$, and (2) a partition $\langle Y_1,\ldots,Y_p\rangle \in Part(\mathbf{x} \setminus Z)$. In other words, we have $[\![\varphi]\!]_{\mathbf{x}}(k) = \bigcup\{[\![\varphi]\!]_Z^{Y_1,\ldots,Y_p} \mid Z \subseteq \mathbf{x} \cup \{x\},\ x \in Z,\ \langle Y_1,\ldots,Y_p\rangle \in Part(\mathbf{x} \setminus Z)\}$, where $[\![\varphi]\!]_Z^{Y_1,\ldots,Y_p}$ is defined in Table 1. Intuitively, Z corresponds to the set of support variables that are aliased with x in some concrete model, and $\langle Y_1,\ldots,Y_p\rangle$ is used with the same meaning as in the previous definition of $[\![\text{emp}]\!]_{\mathbf{x}}(k)$.

We recall upon the fact that the models of the atomic propositions $x \mapsto nil$ and $x \mapsto y$ are all heaps whose domains consists of only one address, which is the value of x. Therefore the SGR representation uses one non-dangling node n to which one counter z_1 is mapped. The associated constraint sets $z_1 = 1$, according to the semantics.

Table 1. SGR for atomic spatial propositions

	$[\![\text{emp}]\!]^{Y_1,\ldots,Y_p}$	$[\![x \mapsto \text{nil}]\!]_Z^{Y_1,\ldots,Y_p}$	$[\![x \mapsto y]\!]_Z^{Y_1,\ldots,Y_p}$	$[\![ls^l(x,\text{nil})]\!]_{Z_1,\ldots,Z_k}^{Y_1,\ldots,Y_p}$	$[\![ls^l(x,y)]\!]_{Z_1,\ldots,Z_k}^{Y_1,\ldots,Y_p}$
N	$D \cup \{Nil\}$	$D \cup \{n, Nil\}$	$D \cup \{n, Nil\}$	$D \cup \{n_1,\ldots,n_k, Nil\}$	$D \cup \{n_1,\ldots,n_k, Nil\}$
Z	$\emptyset$	$\{\langle n, z_1\rangle\}$	$\{\langle n, z_1\rangle\}$	$\{\langle n_i, z_i\rangle\}_{i=1}^k$	$\{\langle n_i, z_i\rangle\}_{i=1}^k$
S	$\emptyset$	$\{\langle n, Nil\rangle\}$	$\{\langle n, d_k\rangle\}$, if $y \in Y_k$	$\{\langle n_i, n_{i+1}\rangle\}_{i=1}^{k-1} \cup \{\langle n_k, Nil\rangle\}$	$\{\langle n_i, n_{i+1}\rangle\}_{i=1}^{k-1}$
V	Δ	$\lambda x : Z.n \cup \Delta$	$\lambda x : Z.n \cup \Delta$	$\bigcup_{i=1}^k \lambda x : Z_i.n_i \cup \Delta$	$\bigcup_{i=1}^k \lambda x : Z_i.n_i \cup \Delta$
φ	$\top$	$z_1 = 1$	$z_1 = 1$	$\sum_{i=1}^k z_k = l \wedge$ $(x \in Z_k \rightarrow l = 0)$	$\sum_{i=1}^k z_k = l \wedge$ $(x, y \in Z_k \rightarrow l = 0)$

In the definition of $[\![ls^l(x,nil)]\!]_{\mathbf{x}}(k)$ we consider an ordered sequence of disjoint subsets of $\mathbf{x}$, namely $Z_1,\ldots,Z_k$, where $Z_i \subseteq \mathbf{x} \cup \{x\}$, $1 \le k \le n$, such that $x \in Z_1$, and $Z_i \cap Z_j = \emptyset$, for all $1 \le i < j \le k$. Similarly, in the definition of $[\![ls^l(x,y)]\!]_{\mathbf{x}}(k)$ we consider sets $Z_1,\ldots,Z_k$, where $Z_i \subseteq \mathbf{x} \cup \{x,y\}$, $1 \le k \le n$, such that $x \in Z_1$, $y \in Z_k$, and $Z_i \cap Z_j = \emptyset$, for all $1 \le i < j \le k$. In both cases, we consider also a partition $\langle Y_1,\ldots,Y_p\rangle \in Part(\mathbf{x} \setminus (\bigcup_{i=1}^k Z_i))$. Intuitively, $Z_1,\ldots,Z_k$ correspond to the sets of support variables that are aliased while pointing to the same node in the list, in some concrete model, and $\langle Y_1,\ldots,Y_p\rangle$ is used with the same meaning as in the previous definition of $[\![\mathrm{emp}]\!]_{\mathbf{x}}(k)$.

Since the models of $ls^k(x,nil)$ and $ls^l(x,y)$ are all heaps defined only on the addresses of the nodes in the list pointed to by x, we represent them by a list of symbolic non-dangling nodes $n_1,\ldots,n_k$, where all variables from Z_i point to n_i, $1 \le i \le k$. Each node n_i has an associated counter z_i, and the sum of the values of z_i must equal l. Moreover, if x points to the end (nil or y) of the list, the length l must be zero.

The pure formulae $x = \mathrm{nil}$ ($x = y$) correspond to sets of SGRs are the ones in which x points to nil (y), and the counters occur unconstrained. Their SGR semantics is defined as follows:

$$\begin{array}{rl}
[\![x = \mathrm{nil}]\!]_{\mathbf{x}}(k) = & \{\langle G,\top\rangle \mid G = \langle N,D,R,Z,S,V\rangle \in \mathcal{S}_k(\mathbf{x}),\ V(x) = Nil\} \\
[\![x = y]\!]_{\mathbf{x}}(k) = & \{\langle G,\top\rangle \mid G = \langle N,D,R,Z,S,V\rangle \in \mathcal{S}_k(\mathbf{x}),\ V(x) = V(y)\}
\end{array}$$

The SGR semantics for the QSL connectives is defined as follows:

$$\begin{array}{rcl}
[\![\mathbb{T}]\!]_{\mathbf{x}}(k) & = & \{\langle G,\top\rangle \mid G \in \mathcal{S}_k(\mathbf{x})\} \\
[\![\psi_1 \wedge \psi_2]\!]_{\mathbf{x}}(k) & = & [\![\psi_1]\!]_{\mathbf{x}}(k) \sqcap [\![\psi_2]\!]_{\mathbf{x}}(k) \\
[\![\neg\psi]\!]_{\mathbf{x}}(k) & = & \{\langle G,\top\rangle \mid G \in \mathcal{S}_k(\mathbf{x})\} \ominus [\![\psi]\!]_{\mathbf{x}}(k) \\
[\![\psi_1 * \psi_2]\!]_{\mathbf{x}}(k) & = & [\![\psi_1]\!]_{\mathbf{x}}(k) \circledast [\![\psi_2]\!]_{\mathbf{x}}(k)
\end{array}$$

If π is a purely arithmetic formula, then we have:

$$[\![\psi \wedge \pi]\!]_{\mathbf{x}}(k) = \{\langle G,\theta \wedge \pi\rangle \mid \langle G,\theta\rangle \in [\![\psi]\!]_{\mathbf{x}}(k)\}$$

The semantics for the existential quantifiers is as follows:

$$\begin{array}{rcl}
[\![\exists x \,.\, \psi]\!]_{\mathbf{x}}(k) & = & \{R\downarrow_x \mid R \in [\![\psi]\!]_{\mathbf{x}\cup\{x\}}(k-1)\},\ k \ge 1 \\
[\![\exists_{\mathbb{N}} l \,.\, \psi]\!]_{\mathbf{x}}(k) & = & \{\langle G,\exists l \,.\, \theta\rangle \mid \langle G,\theta\rangle \in [\![\psi]\!]_{\mathbf{x}}(k)\}
\end{array}$$

The following lemma formalizes the correctness of our construction.

Lemma 6. *Given φ a* **QSL** *formula containing only numeric quantifiers ($\exists_{\mathbb{N}}$, $\forall_{\mathbb{N}}$), and $\{x_1,\ldots,x_n\} \subseteq FV_L(\varphi)$, we have, for all valuations $\nu : FV_L(\varphi) \setminus \{x_1,\ldots,x_n\} \to Loc$, and $\iota : FV_I(\varphi) \to \mathbb{N}^+$: $\Gamma_{\nu,\iota}\big([\![\exists x_1 \ldots \exists x_n \,.\, \varphi]\!]_{FV_L(\varphi)\setminus\{x_1,\ldots,x_n\}}(n)\big) = \{H \mid \langle H,\nu,\iota\rangle \models \exists x_1 \ldots \exists x_n \,.\, \varphi\}$*

Theorem 2. *The validity of entailments between formulae in the $\exists^*\{\exists_{\mathbb{N}},\forall_{\mathbb{N}}\}^*$ quantifier fragment of* **QSL** *is a decidable problem.*

The proof of Theorem 2 uses the fact that the set of models for each formula in the $\exists^*\{\exists_{\mathbb{N}},\forall_{\mathbb{N}}\}^*$ quantifier fragment of **QSL** can be finitely represented using a set of SGRs (cf. Lemma 6). An entailment $\varphi \Rightarrow \psi$ is valid if and only if the set of models of the formula $\varphi \wedge \neg\psi$ is empty. The latter is given by the set of SGRs encoding the set-theoretic difference between the models of φ and the models of ψ. Since the emptiness of this set is decidable, by reduction to Presburger arithmetic, the validity of the entailment is also decidable.

According to Lemma 1, the number of SSGs that can be generated using N variables is of the order of $O(N^N)$. However, the number of SSGs encountered in practice is relatively small, since in principle, the explosion occurs only due to the unrestricted use of negation and the $\mathbb{T}$ proposition, which can be easily avoided.

5 Application of the Model Theoretic Method for QSL

The translation between **QSL** formulae with quantifier prefix of the form $\exists^*\{\exists_{\mathbb{N}}, \forall_{\mathbb{N}}\}^*$ and sets of SGRs gives a method for deciding the validity of entailments in this logic. Moreover, there is another, more practical advantage to this approach, that gives us a effective method for the verification of both shape and numeric properties of programs with lists.

The L2CA tool [3] is a tool for verifying safety and termination properties of programs with singly-linked lists, based on the translation of programs into counter automata [9]. A counter automaton generated by L2CA has control states of the form $\langle l, G\rangle$, where l is a control label of the original program, and G is a SSG over the set *PVar* of pointer variables of the input program. By Lemma 1, the set of control states of a counter automaton generated by L2CA is finite, which guarantees that each program with lists will be translated into a finite-control counter automaton. The semantics (set of runs) of the counter automaton generated by L2CA is in a bisimulation relation with the semantics of the original program, therefore all results of the analysis of the counter automaton (e.g. safety properties, termination) carry over to the original program.

The fact that any $\exists^*\{\exists_{\mathbb{N}}, \forall_{\mathbb{N}}\}^*$ **QSL** formula φ corresponds to a set $[\![\varphi]\!]$ of pairs $\langle G, \psi\rangle$, where G is an SSG and ψ is a Presburger constraint, allows us to extend the L2CA tool to check total correctness of Hoare triples in which the pre- and post-conditions are expressed as $\exists^*\{\exists_{\mathbb{N}}, \forall_{\mathbb{N}}\}^*$ **QSL** formulae. Suppose that $\{\varphi\}$ **P** $\{\psi\}$ is such a triple. Then for each SGR $\langle G_k, \phi_k\rangle \in [\![\varphi]\!]$ the L2CA tool will generate a counter automaton A_k with initial state $\langle l_0, G_k\rangle$, where l_0 is the initial control label of the program **P**. This automaton corresponds to the semantics of **P** when started in an initial control state $\langle l_0, H_0\rangle$, where $H_0 \in \Gamma(\langle G_k, \phi_k\rangle)$. Let A be the union of all such A_k.

By using a combination of existing tools for the analysis of counter automata, e.g. [6,2,1] we can verify whether A, started in each control state $\langle l_0, G_k\rangle$ with values of counters satisfying the Presburger constraint ϕ_k, reaches a final control state $\langle l_f, G_f\rangle$ with the counters satisfying some Presburger constraint ϕ[1] such that $\models \phi \rightarrow \phi'$, for some $\langle G_f, \phi'\rangle \in [\![\psi]\!]$. This suffices for checking partial correctness. On what concerns total correctness, we use a termination analysis tool for counter automata, e.g. [1], to check whether **P**, started with any heap H_0 such that $H_0 \models \varphi$, terminates.

[1] In general this is an over-approximation of the set of reachable configurations, obtained using a combination of precise (acceleration) and abstract (widening) methods.

5.1 Experimental Results

Table 2 presents some experimental results of verifying Hoare triples of the form $\{\varphi\}$ **P** $\{\psi\}$, where φ and ψ are **QSL** formulae, and **P** is a program handling lists. The ListReversal example receives in input a non-circular list pointed to by u of length l and returns a non-circular list pointed to by v containing the cells of the first list in reversed order. The BubbleSort and InsertSort programs are classical sorting algorithms for which we verified that the length of the input list stays the same. The ListCounter example is a simple loop traversing a list pointed to by u, while incrementing an integer counter c. InsertDelete is the example from Figure 2.

Table 2. Experimental Results using the L2CA and ASPIC tools

$\{\varphi\}$	**P**	$\{\psi\}$	Size	Gen (s)	Verif (s)	Tool
$\{ls^l(u,nil)\}$	ListReversal	$\{ls^l(v,nil)\}$	4	0.4	0.3	Fast
$\{ls^l(u,nil)\}$	BubbleSort	$\{ls^l(u,nil)\}$	25	0.4	0.4	Aspic
$\{ls^l(u,nil)\}$	InsertSort	$\{ls^l(u,nil)\}$	58	0.6	0.6	Aspic
$\{ls^l(u,nil) \wedge c = 0\}$	ListCounter	$\{ls^l(u,nil) \wedge c = l\}$	16	0.2	0.1	Aspic
$\{c = 0 \wedge emp \wedge u = nil\}$	InsertDelete	$\{c = 0 \wedge emp\}$	6	1.5	0.5	Fast

For all examples, the size (number of control locations) of the automata generated by L2CA is given in the second (Size) column, the time needed for generation in the third (Gen) column, and the time needed to verify partial correctness of the model is given in the fourth (Verif) column. The tool used (either Aspic [2] or Fast [6]) is given in the fifth column. All programs were found to be correct.

6 Conclusions

We have developed an extension of Separation Logic interpreted over singly-linked heaps, that allows to specify properties related to the sizes of the lists. This logic is especially useful for reasoning about programs that combine dynamically allocated data with variables ranging over integer domains.

The decidability of the extended logic is studied, the full quantifier fragment being shown to be undecidable, by a reduction from the halting problem for 2 counter machines. However the validity of entailments in the $\exists^*\{\exists_{\mathbb{N}}, \forall_{\mathbb{N}}\}^*$ fragment of the logic is decidable, which allows the use this fragment to specify Hoare triples for programs with lists. The verification of total correctness properties specified in this way was made possible by an extension of the L2CA tool.

References

1. ARMC, http://www.mpi-sb.mpg.de/~rybal/armc/
2. ASPIC, http://www-verimag.imag.fr/~gonnord/aspic/aspic.html
3. L2CA, http://www-verimag.imag.fr/~async/L2CA/l2ca.html

4. Smallfoot, http://www.dcs.qmul.ac.uk/research/logic/theory/projects/smallfoot/index.html
5. Annichini, A., Bouajjani, A., Sighireanu, M.: Trex: A tool for reachability analysis of complex systems. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 368–372. Springer, Heidelberg (2001)
6. Bardin, S., Finkel, A., Leroux, J., Petrucci, L.: Fast: Fast accelereation of symbolic transition systems. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988. Springer, Heidelberg (2004)
7. Benedikt, M., Reps, T., Sagiv, M.: A decidable logic for describing linked data structures. In: Swierstra, S.D. (ed.) ESOP 1999. LNCS, vol. 1576. Springer, Heidelberg (1999)
8. Berdine, J., Calcagno, C., O'Hearn, P.: A Decidable Fragment of Separation Logic. In: Lodaya, K., Mahajan, M. (eds.) FSTTCS 2004. LNCS, vol. 3328. Springer, Heidelberg (2004)
9. Bouajjani, A., Bozga, M., Habermehl, P., Iosif, R., Moro, P., Vojnar, T.: Programs with lists are counter automata. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144. Springer, Heidelberg (2006)
10. Bozga, M., Iosif, R., Perarnau, S.: Quantitative separation logic and programs with lists. Technical Report TR 2007-9, VERIMAG (2007)
11. Burstall, R.M.: Some techniques for proving correctness of programs which alter data structures. Machine Intelligence 7, 23–50 (1972)
12. Gulwani, S., McCloskey, B., Tiwari, A.: Lifting abstract interpreters to quantified logical domains. In: Proc. 35th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. ACM Press, New York (2008)
13. Gulwani, S., Tiwari, A.: An abstract domain for analyzing heap-manipulating low-level software. In: Proc. Intl. Conference on Computer Aided Verification (2007)
14. Immerman, N., Rabinovich, A., Reps, T., Sagiv, M., Yorsh, G.: Verification via Structure Simulation. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114. Springer, Heidelberg (2004)
15. Ishtiaq, S., O'Hearn, P.: BI as an assertion language for mutable data structures. In: POPL (2001)
16. Klaedtke, F., Ruess, H.: Monadic second-order logics with cardinalities. In: Proc. 30th International Colloquium on Automata, Languages and Programming. LNCS. Springer, Heidelberg (2003)
17. Magill, S., Berdine, J., Clarke, E., Cook, B.: Arithmetic Strengthening for Shape Analysis. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634. Springer, Heidelberg (2007)
18. Minsky, M.: Computation: Finite and Infinite Machines. Prentice-Hall, Englewood Cliffs (1967)
19. O'Hearn, P., Calcagno, C., Yang, H.: Computability and Complexity Results for a Spatial Assertion Language for Data Structures. In: Hariharan, R., Mukund, M., Vinay, V. (eds.) FSTTCS 2001. LNCS, vol. 2245. Springer, Heidelberg (2001)
20. Presburger, M.: Über die Vollstandigkeit eines gewissen Systems der Arithmetik. Comptes rendus du I Congrés des Pays Slaves, Warsaw (1929)
21. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proc. 17th IEEE Symposium on Logic in Computer Science. LNCS. Springer, Heidelberg (2002)
22. Wolper, P., Boigelot, B.: Verifying systems with infinite but regular state spaces. In: Y. Vardi, M. (ed.) CAV 1998. LNCS, vol. 1427, pp. 88–97. Springer, Heidelberg (1998)
23. Yorsh, G., Rabinovich, A., Sagiv, M., Meyer, A., Bouajjani, A.: A logic of reachable patterns in linked data-structures. In: Proc. Foundations of Software Science and Computation Structures. LNCS. Springer, Heidelberg (2006)
24. Zhang, T., Sipma, H., Manna, Z.: Decision procedures for recursive data structures with integer constraints. In: Proc. Intl. Joint Conference of Automated Reasoning (2004)

On Automating the Calculus of Relations

Peter Höfner[1] and Georg Struth[2]

[1] Institut für Informatik, Universität Augsburg, Germany
hoefner@informatik.uni-augsburg.de
[2] Department of Computer Science, University of Sheffield, United Kingdom
g.struth@dcs.shef.ac.uk

Abstract. Relation algebras provide abstract equational axioms for the calculus of binary relations. They name an established area of mathematics and have found numerous applications in computing. We prove more than hundred theorems of relation algebras with off-the-shelf automated theorem provers. They form a basic calculus from which more advanced applications can be explored. We also present two automation experiments from the formal methods literature. Our results further demonstrate the feasibility of automated deduction with complex algebraic structures. They also open a new perspective for automated deduction in relational formal methods.

1 Introduction

Relations are among the most ubiquitous concepts in mathematics and computing. Relational calculi have their origin in the late nineteenth century and their initial development was strongly influenced, but then overshadowed, by the advancement of mathematical logic. Around 1940, Alfred Tarski revived the subject by formalising the calculus of binary relations alternatively within the three-variable fragment of first-order logic and as an abstract relation algebra within first-order equational logic [28]. Today, relation algebras form an established field of mathematics with numerous textbooks and research publications.

The relevance of relational calculi in computing has been realised since the early beginnings. Relational approaches had considerable impact on program semantics, refinement and verification through the work of Dijkstra, Hoare, Scott, de Bakker, Back and others. Formal methods like Alloy [14], B [2] and Z [25], or Bird and de Moor's algebraic approach to functional program development [6] are strongly relational. Further applications of relations in computing include data bases, graphs, preference modelling, modal reasoning, linguistics, hardware verification and the design of algorithms. Here, our main motivation is program development and verification.

To support relational formal methods, various tools have been developed. Interactive proof-checkers for relation algebras have been implemented [30,16] and relational techniques have been integrated into various proof checkers for B or Z. Special purpose first-order proof systems for relation algebras, including tableaux and Rasiowa-Sikorski calculi, have been proposed [18,17]. Translations of relational expressions into (undecidable) fragments of predicate logics have been implemented [24] and integrated into the SPASS theorem prover [31]. Finite relational properties can efficiently be analysed with tools similar to model checkers [4]. Modal logics can automatically be translated into relational calculi to be further analysed by the relational tools available [10]. But is it really necessary to base automated reasoning with relation algebras either on interactive theorem provers, on special-purpose calculi or on finitary methods? Would off-the-shelf automated theorem provers (APT systems) necessarily fail to derive anything interesting when provided with Tarski's equational axioms?

This paper provides the proof of concept that a direct axiomatic integration of relation algebras into modern off-the-shelf ATP systems is indeed feasible. Our main contributions are as follows: First, we identify axiomatisations of relation algebras that are particularly suitable for proof search. Second, we prove more than hundred theorems of relation algebra with the ATP system Prover9 and the counterexample generator Mace4 [21]. These experiments mechanise the calculus from a standard mathematical textbook [20] and yield a verified basis on which further applications can be built. Experiments show that Prover9 and Waldmeister are currently best suited for this task [8]. Third, we present two extended examples that further demonstrate the applicability of the approach. The first one automates an example from Abrial's B-Book [2] that analyses kinship relations in a fictitious society. The second one automatically analyses simulation laws in data refinement. In addition, we consider relation algebras as Boolean algebras with operators, thus take the initial steps towards reasoning automatically about modalities in this approach. Detailed information, including all input and output files (in TPTP format) can be found at a website [1]. Selected theorems will become part of the TPTP library in summer 2008 [26].

Our overall experience is positive: Many textbook-level theorems and calculational proofs that eminent mathematicians found worth publishing some decades ago can nowadays be automatically verified within a few minutes from the axioms of relation algebras by off-the-shelf ATP systems. More complex statements require either inequational reasoning, for which we also provide an axiomatisation, or "learning" of the hypotheses, for which we use heuristics. In conclusion, the axiomatic integration of computational algebras into off-the-shelf ATP systems offers a simple yet powerful alternative to interactive approaches, special-purpose procedures and finitary methods. A new kind of application of automated deduction in formal methods seem therefore possible.

2 Binary Relations and Relation Algebras

Binary relations and their basic operations feature in most introductory courses on discrete mathematics. More information can be found, e.g., in the textbooks by Maddux [19], and Schmidt and Ströhlein [23].

A binary relation R on a set A is just a subset of $A \times A$ — a set of ordered pairs. Since relations are sets, unions $R \cup S$, intersections $R \cap S$ and complements $\overline{R}$ of relations can be taken such that the set of all binary relations forms a Boolean algebra. The *relative product* $R;S$ of two relations R and S is the set of all pairs (a,b) such that $(a,c) \in R$ and $(c,b) \in S$ for some $c \in A$. The *converse* $\breve{R}$ of a relation R is the set of all pairs (a,b) with $(b,a) \in R$. The *identity relation* 1_A on A is the set of all pairs (a,a) with $a \in A$. The structure $(2^{A^2}, \cup, ;, \overline{\ }, \breve{\ }, 1_A)$ is called proper relation algebra of all binary relations over A. To define relation algebras more abstractly, binary relations are replaced by arbitrary elements of some carrier set A and a set of equational axioms is given.

A *relation algebra* is a structure $(A, +, ;, \overline{\ }, \breve{\ }, 1)$ satisfying the axioms

$$\begin{gathered}(x+y)+z = x+(y+z)\ , \qquad x+y = y+x\ , \qquad x = \overline{\overline{x}+\overline{y}} + \overline{\overline{x}+y}\ , \\ (x;y);z = x;(y;z)\ , \qquad (x+y);z = x;z+y;z\ , \qquad x;1 = x\ , \\ \breve{\breve{x}} = x\ , \qquad (x+y)\breve{} = \breve{x}+\breve{y}\ , \qquad \breve{x};\overline{x;y}+\overline{y} = \overline{y}\ .\end{gathered}$$

We assume that relative products bind more strongly than joins and meets and that complementation and converse bind more strongly than relative products. The first line contains Huntington's axioms for Boolean algebras [13,12]. The join operation is denoted by $+$, and the meet of Boolean algebra can be defined as $x \cdot y = \overline{\overline{x}+\overline{y}}$. Since relation algebras are Boolean algebras, they form posets with respect to $x \le y \Leftrightarrow x+y = y$ that have greatest elements $\top = x+\overline{x}$ and least elements $0 = x \cdot \overline{x}$. The axioms in the second line define the relative product. The axioms in the third line define the operation of conversion. A TPTP-encoding is given in Appendix A.

On the one hand, this axiomatisation is very compact and initial experiments suggest that it is particularly suitable for automation. Further hypotheses can easily be added by need while keeping the initial set small. On the other hand, humans may find it difficult to prove even simple facts from these axioms alone.

A relation algebra is representable iff it is isomorphic to a proper relation algebra, but not all relation algebras have that property. This means that they are too weak to prove some truths about binary relations. But this weakness is balanced by a strength: While the expressiveness of the calculus of binary relations is precisely that of the three-variable fragment of first-order logic, a translation into logic can introduce quite complex expressions with nested quantifiers and destroy the inherent algebraic structure of a statement. This can obfuscate the decomposition of complex theorems into lemmas and the control of hypotheses needed in proofs. Relation algebras can yield simpler, more modular and more concise specifications and proofs. This situation is similar to pointwise versus pointfree functional programming. Relation algebras are also interesting for ATP systems because already the equational theory is undecidable.

3 Boolean Algebras: A Warm-Up

The automated analysis of Boolean algebras is one of the great success stories of automated deduction [22]; proofs with these and similar lattices are well-documented through the TPTP library and various publications. Here and in the next sections we follow Maddux's book [19] in proving a series of theorems that constitute the core calculus of relation algebras. The series starts with proving Boolean properties from Huntington's axioms that are later useful in relational proofs. All experiments used a Pentium 4 processor with 3 GHz and Hyper-Threading and 2 GB memory. All input files in TPTP-format and all Prover9 outputs can be found at a website [1]. Here, we only briefly summarise our results. As a default, we proved our goals from the full axiom set. All deviations are explicitly mentioned.

Proposition 3.1. *Huntington's axioms imply the standard equational axioms for Boolean algebras.*

By "standard" we mean the usual equational lattice axioms, the distributivity law(s) and the axioms for Boolean complementation that can be found in any book on lattices. In particular, Prover9 showed in less than one second that $x + \overline{x} = y + \overline{y}$ and $x \cdot \overline{x} = y \cdot \overline{y}$. This expresses the existence of a unique greatest element $\top$ and a unique least element 0 such that $x + \overline{x} = \top$ and $x \cdot \overline{x} = 0$.

Lattices, Boolean algebras, and therefore relation algebras can also be defined as posets, and order-based proofs by hand are usually easier. A proof of $x = y$ can be achieved by proving $x \leq y$ and $y \leq x$ separately. We therefore provide an order-based encoding as an alternative to the equational one by adding the definition $x \leq y \Leftrightarrow x + y = y$. Join, meet, relative product and conversion are isotone with respect to $\leq$; complementation is antitone. For example,

$$x \leq y \Rightarrow z;x \leq z;y \quad \text{and} \quad x \leq y \Rightarrow \overline{y} \leq \overline{x} \ .$$

Reflexivity and transitivity of $\leq$ and the monotonicity properties can be very useful additional hypotheses in more complex examples. Our general experience is that order-based automated proofs are better when there is a simple order-based proof by hand. But there are exceptions to the rule and in advanced applications, both variants should be explored.

Proposition 3.2. *Huntington's axioms and the order-based axioms for Boolean algebras are equivalent.*

Prover9 needed less than one second for the left-to-right direction and less than two minutes for its converse.

In sum, we proved more than 40 theorems about Boolean algebras (the TPTP-library currently contains around 100). Prover9 succeeded with every single task we tried. This basic library of automatically verified statements can be used as hypotheses for more advanced applications with relation algebras.

4 Boolean Algebras with Operators

Boolean algebras provide the foundations for more complex structures such as relation algebras, cylindric algebras [11] or modal algebras [7]. All these structures can be understood as Boolean algebras with operators, which makes them interesting candidates for ATP. But to our knowledge, an axiom-based approach with off-the-shelf ATP systems has not yet been attempted. This section further follows Maddux's book, but also Jónsson and Tarski's seminal article on Boolean algebras with operators [15] by automatically proving some of their calculational statements. Since the proofs in these sources are essentially order-based, we strongly rely on that axiomatisation for Prover9.

According to a classical definition by Jónsson and Tarski, two functions f and g on a Boolean algebra are *conjugate* if they satisfy

$$f(x) \cdot y = 0 \Leftrightarrow x \cdot g(y) = 0 \ .$$

With the order-based encoding of Boolean algebras and the conjugation property, Prover9 could verify a series of laws that is documented in Table 1. Again, the most important laws can be organised into a lemma.

Table 1. Laws for Conjugates

#	Theorem	t[s]	#	Theorem	t[s]
(1)	$f(x+y) \leq z \Leftrightarrow f(x)+f(y) \leq z$	182.51	(2)	$f(x+y) = f(x)+f(y)$	0.16
(3)	$f(0) = 0$	0.10	(4)	$x \leq y \Rightarrow f(x) \leq f(y)$	14.98
(5)	$f(\overline{g(x)}) \leq \overline{x}$	138.93	(6)	$f(x \cdot y) \leq f(x) \cdot f(y)$	147.64
(7)	$f(x)+f(y) \leq f(x+y)$	141.11	(8)	$f(x) \leq y \Rightarrow x \leq \overline{g(\overline{y})}$	34.81
(9)	$f(x) \leq y \Leftarrow x \leq \overline{g(\overline{y})}$	8.10	(10)	$f(x) \cdot y \leq f(x \cdot g(y)) \cdot y$	241.92
(11)	$\forall x, y.(f(x) \cdot y = 0 \Leftrightarrow x \cdot h(y) = 0) \Rightarrow \forall z.(g(z) = h(z))$				86.75
(12)	$f(x \cdot \overline{g(y)}) \leq f(x) \cdot \overline{y}$	144.81			

Lemma 4.1. *Conjugate functions on a Boolean algebra*

(i) are strict and additive;
(ii) induce a Galois connection;
(iii) satisfy modular laws;
(iv) are in one-to-one correspondence.

Proof. (i) holds by Equation (1), (2) and (3); (ii) holds by Equation (8) and (9); (iii) holds by Equation (10); (iv) holds by Equation (11). □

Often, a law of the form $f(x) \cdot y \leq f(x \cdot g(y))$ is called *modular law*. Other properties displayed in Table 1 are isotonicity of conjugates (Equation (4)), a cancellation law (Equation (5)) and a subdistributivity law (Equation (6)).

Most of the theorems could be proved entirely from Huntington's axioms and the conjugation axiom, but some more complex ones such as (10) required the addition of additional hypotheses, like the standard distributivity laws.

We used a manual form of "hypothesis learning" which can easily be implemented as an automated procedure. To this end, we started with very small axiom sets from which "explosive" axioms such as commutativity had been discarded.

We then added further hypotheses until a counterexample generator failed and there was hope that the hypotheses are strong enough to entail the goal.The selection of these hypotheses was usually based on a mixture of semantic knowledge and blind guessing. We then tried the ATP system and repeated the procedure with different hypotheses if the proof search failed within reasonable time. Usually, we set a time limit of 300 s. It has been experimentally confirmed that the probability that an ATP system will prove a theorem beyond that threshold is very low [27]. Interestingly, we frequently encountered situations where hypotheses that were crucial for success were not needed in the proof itself, but acted as catalysers for redundancy elimination. Detailed information about all deviations from the standard axiomatisation is provided at our website [1].

The results of this section make some proofs in relation algebras (e.g., the modular laws) simpler and more convenient. But they are also of wider interest. Boolean algebras with operators are algebraic variants of (multi)modal logics. Our experiments therefore suggest that the automation of modal logics through the combination of off-the-shelf ATP systems with algebras might be a feasible alternative to existing special-purpose calculi and decision procedures that extends to undecidable first-order modal logics.

5 Relation Algebras

Relation algebras can be perceived as Boolean algebras with operators corresponding to functions like $\lambda x.a;x$. As already mentioned, we follow Maddux's first-order axiomatisation. There are second-order variants of relation algebras in which the underlying Boolean algebra is assumed to be complete and atomic [23]. Relation algebras are quite rich and complex structures; they are expressive enough for modelling set theory [29]. Again we can group some of our experiments into lemmas.

Lemma 5.1. *Relation algebras are idempotent semirings with respect to join and composition.*

Proof. An idempotent semiring is a structure $(S,+,;,0,1)$ such that $(S,+,0)$ is a commutative idempotent monoid, $(S,;,1)$ is a monoid, multiplication distributes over addition from left and right, and 0 is a left and right annihilator, i.e., $0;a = 0 = a;0$. The facts needed for proving this are shown in Table 2. □

Table 2. Relational Semiring Laws

#	Theorem	t[s]
(13)	$x;(y+z) = x;y + x;z$	3.9
(14)	$1;x = x$	0.06
(15)	$0;x = 0$	
(16)	$x;0 = 0$	

Table 3. Isotonicity Laws

#	Theorem	t[s]
(17)	$x \leq y \Leftrightarrow \breve{x} \leq \breve{y}$	0.13
(18)	$x \leq y \Rightarrow x; z \leq y; z$	100.31
(19)	$x \leq y \Rightarrow z; x \leq z; y$	

Table 4. Schröder Laws

#	Theorem	t[s]
(20)	$x; y \cdot z = 0 \Rightarrow y \cdot \breve{x}; z = 0$	287.82
(21)	$x; y \cdot z = 0 \Leftarrow y \cdot \breve{x}; z = 0$	264.49

Lemma 5.2. *In relation algebras,*

(i) relative products and conversion are isotone;
(ii) the maps $\lambda y.x; y$ and $\lambda y.\breve{x}; y$ are conjugates.

Proof. See Table 3 for (i) and Table 4 for (ii). □

The equivalence expressed by Equations (20) and (21) is called *Schröder law.* This law is one of the working horses of relation algebras; often in the equivalent and the dual form $x; y \leq \overline{z} \Leftrightarrow \breve{x}; z \leq \overline{y} \Leftrightarrow z; \breve{y} \leq \overline{x}$. The Schröder laws are of course also very helpful additional hypotheses for ATP systems.

The fact that the Schröder laws express conjugation shows a significant limitation of the first-order approach. In a higher-order setting it would now be possible to transfer all generic properties of conjugate functions or adjoints of a Galois connection to the relational level. It would also be possible to exploit the semiring duality that links the two equivalences of the Schröder laws. In the pure first-order setting, all this work remains explicit.

Having identified the above conjugation it is evident that modular laws hold in relation algebras, too. But while an automation in Boolean algebra with operators was possible, we did not succeed in relation algebra without an axiom restriction. The reason is that the operation of function application, which is present in Boolean algebras with operators but not in relation algebras, reduces the applicability of associativity of composition and prunes the search space. To learn the appropriate restriction, we reused the axioms listed in the Prover9 output for Boolean algebras with operators. This was the key to success.

Lemma 5.3. *The modular laws and the Dedekind law hold in relation algebras.*

Proof. See Table 5. □

Table 5. Modular and Dedekind Laws

#	Theorem	t[s]
(22)	$x; (y \cdot z) \leq x; y \cdot x; z$	25.34
(23)	$z; x \cdot y \leq z; (x \cdot \breve{z}; y) \cdot y$	5444.61
(24)	$z; x \cdot y \leq (z \cdot y; \breve{x}); (x \cdot \breve{z}; y)$	3.28

The Dedekind law (24) is another fundamental law of relation algebras. It is also the most complex law automated in this section. We had to restrict the relation algebra axioms and to add the modular laws as further hypotheses.

A rich calculus can be developed from these basic laws. Most of the laws we tried could again be proved without any restriction from the relation algebra axioms and with reasonable running times. Examples are displayed in Table 6.

Table 6. Further Relational Laws

#	Theorem	t[s]
(25)	$\breve{0} = 0$	0.35
(26)	$\breve{\top} = \top$	
(27)	$\breve{\overline{x}} = \overline{\breve{x}}$	
(28)	$(x \cdot y)\breve{} = \breve{x} \cdot \breve{y}$	
(29)	$0 = \breve{x} \cdot y \Leftrightarrow 0 = x \cdot \breve{y}$	0.45
(30)	$\breve{1} = 1$	0.03
(31)	$x \leq x; \top$	0.26
(32)	$x \leq \top; x$	
(33)	$\top; \top = \top$	
(34)	$x; y \cdot \overline{x; z} = x; (y \cdot \overline{z}) \cdot \overline{x; z}$	184.71
(35)	$\overline{y; x}; \breve{x} \leq \overline{y}$	3.81

Our experiments show that the calculus of relation algebras, as presented in textbooks, can be automated without major obstacles. This does not mean that this calculus is trivial. Novices might find it difficult to prove the laws in this section by hand from the axioms given.

6 Functions, Vectors and Other Concepts

Relation algebras allow the abstract definition of various concepts, including functions, vectors, points, residuals, symmetric quotients or subidentities, and the proof of their essential properties [19,23]. These and more advanced concepts are important, for instance, for program development with Alloy, B or Z, or for the construction and verification of functional programs. Abrial's B-Book [2], in particular, contains long lists of algebraic properties involving these concepts that have been abstracted from concrete binary relations.

A *vector* (or *subset*) is an element x of a relation algebra that satisfies $x; \top = x$. An intuition can perhaps best be provided through finite relations. These can be represented as Boolean matrices with ones denoting that elements corresponding to rows and columns are ordered pairs. In this setting, vectors correspond to row-constant matrices, which are the only matrices that are preserved under multiplication with the matrix that contains only ones. Prover9 could easily verify a series of basic properties of vectors. They are displayed in Table 7. To speed up proofs we sometimes added some natural hypotheses like monotonicities. Again, all details can be found at our website [1].

For proving (41) and (42) we also used the Dedekind law; the other statements did not require further hypotheses.

A *test* (or subidentity) is an element below 1. Prover9 could prove a series of laws for tests that are displayed in Table 8. They immediately imply that the subalgebra of subidentities of a relation algebra is a Boolean algebra.

Table 7. Vector Laws for Relation Algebras

#	Theorem	t[s]
(36)	$x;\top = x \Rightarrow \overline{x};\top = \overline{x}$	0.14
(37)	$x;\top = x \ \&\ y;\top = y \Rightarrow (x \cdot y);\top = x \cdot y$	0.02
(38)	$x;\top = x \Rightarrow (x \cdot 1);y = x \cdot y$	0.13
(39)	$x;\top = x \Rightarrow (y \cdot \breve{x});(x \cdot z) \leq y;(x \cdot z)$	0.22
(40)	$x;\top = x \Rightarrow (y \cdot \breve{x});(x \cdot z) \leq (y \cdot \breve{x});z$	0.27
(41)	$x;\top = x \Rightarrow (y \cdot \breve{x});(x \cdot z) \geq y;(x \cdot z)$	1.46
(42)	$x;\top = x \Rightarrow (y \cdot \breve{x});(x \cdot z) \geq (y \cdot \breve{x});z$	1.61

Table 8. Test Laws for Relation Algebras

#	Theorem	t[s]
(43)	$x \leq 1 \Rightarrow \breve{x} = x$	15.26
(44)	$x \leq 1 \Rightarrow x;\top \cdot y = x;y$	63.38
(45)	$x \leq 1 \Rightarrow \overline{x;\top} \cdot 1 = \overline{x} \cdot 1$	15.22
(46)	$x \leq 1 \ \&\ y \leq 1 \Rightarrow x;y = x \cdot y$	8.84
(47)	$x \leq 1 \ \&\ y \leq 1 \Rightarrow x;z \cdot y;z = (x \cdot y);z$	129.94
(48)	$x \leq 1 \Rightarrow x;y \cdot \overline{z} = x;y \cdot \overline{x;z}$	63.71

A (partial) *function* is an element x of a relation algebra satisfying $\breve{x};x \leq 1$. This condition concisely expresses that no domain element of a function can be mapped to more than one range element. Facts about functions are displayed in Table 9. Equation (49) says that functions are closed under composition. In Equation (51) we used the distributivity law (50) to reduce waiting time. Additional experiments with functions are again presented at our website.

An element of a relation algebra is *total* if $1 \leq x;\breve{x}$. It is *injective* if its converse is a function and it is *surjective* if its converse is total. Simple properties like the

Table 9. Laws for Functions

#	Theorem	t[s]
(49)	$\breve{x};x \le 1 \;\&\; \breve{y};y \le 1 \Rightarrow (x;y)\breve{}\,;x;y \le 1$	11.18
(50)	$\breve{x};x \le 1 \Rightarrow x;(y \cdot z) = x;y \cdot x;z$	740.08
(51)	$\breve{x};x \le 1 \Rightarrow x;y \cdot x;\overline{y} = 0$	0.15
(52)	$x \le 1 \Rightarrow \breve{x};x \le 1$	0.01

ones in Table 9 can again easily be automated. Basic results for other entities and derived operations are also feasible.

Finally, it can be verified that the Tarski rule $x \neq 0 \Leftrightarrow \top;x;\top = \top$ is not implied by Maddux's axiomatisation of relation algebra. Mace4 produces a counterexample with 4 elements within a few seconds.

The experiments of this section show that properties of many standard mathematical concepts can easily be proved automatically from our relational basis. Including the proofs in Boolean algebras, our website documents more than 100 theorems about relation algebras.

7 Abrial's Relatives

This section contains a first example that further demonstrates the power of relation algebra combined with state-of-the-art ATP. Abrial's B-Book [2] contains a nice non-programming application of relational reasoning by analysing kinship relations. To make the example more interesting (and less realistic), he imposes some severe restrictions on marriages and families. Abrial's example is appealing because its specification is compact, all relational operations occur, and proofs are short, but non-trivial for humans. The following list shows Abrial's initial

Table 10. Kinship Relations

#	Theorem	t[s]
(53)	$\mathsf{Mother} = \mathsf{Father};\mathsf{Wife}$	1.54
(54)	$\mathsf{Spouse} = \mathsf{Spouse}^{\smile}$	
(55)	$\mathsf{Sibling} = \mathsf{Sibling}^{\smile}$	
(56)	$\mathsf{SiblingInLaw} = \mathsf{SiblingInLaw}^{\smile}$	
(57)	$\mathsf{Cousin} = \mathsf{Cousin}^{\smile}$	
(58)	$\mathsf{Father};\mathsf{Father}^{\smile} = \mathsf{Mother};\mathsf{Mother}^{\smile}$	
(59)	$\mathsf{Father};\mathsf{Mother}^{\smile} = 0$	
(60)	$\mathsf{Mother};\mathsf{Father}^{\smile} = 0$	
(61)	$\mathsf{Father};\mathsf{Children} = \mathsf{Mother};\mathsf{Children}$	1.48

assumptions on kinship relations, but we formalise them in a slightly different, more pointfree way.

1. PERSON is a set (cf. Section 6): $\mathsf{PERSON}; \top = \mathsf{PERSON}$.
2. No person can be a man and a woman at the same time: $\mathsf{Men} \cdot \mathsf{Women} = 0$.
3. Every person is either a man or a woman: $\mathsf{Men} + \mathsf{Women} = \mathsf{PERSON}$.
4. Only women have husbands, who must be men:
   $\overline{\mathsf{Women}}; \mathsf{Husband} = 0 \ \wedge\ \mathsf{Husband}; \overline{\mathsf{Men}} = 0$.
5. Women have at most one husband: $\mathsf{injective}(\mathsf{Husband})$.
6. Men have at most one wife: $\mathsf{Wife} = \mathsf{Husband}^{\smile} \ \wedge\ \mathsf{injective}(\mathsf{Wife})$.
7. Mothers are married women: $\mathsf{Mother} \leq \mathsf{Women} \ \wedge\ \mathsf{Mother}; \overline{\mathsf{Husband}} = 0$.

Abrial then defines further concepts from the ones just introduced.

$$
\begin{aligned}
\mathsf{Spouse} &= \mathsf{Husband} + \mathsf{Wife}\ ,\\
\mathsf{Father} &= \mathsf{Mother}; \mathsf{Husband}\ ,\\
\mathsf{Children} &= (\mathsf{Mother} + \mathsf{Father})^{\smile}\ ,\\
\mathsf{Daughter} &= \mathsf{Children}; \mathsf{Women}\ ,\\
\mathsf{Sibling} &= (\mathsf{Children}^{\smile}; \mathsf{Children}) \cdot \overline{1}\ ,\\
\mathsf{Brother} &= \mathsf{Sibling}; \overline{\mathsf{Women}}\ ,\\
\mathsf{SiblingInLaw} &= \mathsf{Sibling}; \mathsf{Spouse} + \mathsf{Spouse}; \mathsf{Sibling} + \mathsf{Spouse}; \mathsf{Sibling}; \mathsf{Spouse}\ ,\\
\mathsf{NephewOrNiece} &= (\mathsf{Sibling} + \mathsf{SiblingInLaw}); \mathsf{Children}\ ,\\
\mathsf{UncleOrAunt} &= \mathsf{NephewOrNiece}^{\smile}\ ,\\
\mathsf{Cousin} &= \mathsf{UncleOrAunt}; \mathsf{Children}\ .
\end{aligned}
$$

The specification of $\mathsf{Sibling}$, in particular, may deserve some explanation. The relation $\mathsf{Children}^{\smile}; \mathsf{Children}$ links each child not only with its siblings, but also with itself. Intersecting with $\overline{1}$ eliminates the reflexive part of this relation and thus yields the real siblings.

Based on this specification, Abrial presents ten proof tasks which are shown in Table 10, except for a pointwise law the proof of which would require additional axioms. Proofs of the last four facts from Table 10 are displayed in the B-Book and they alone cover more than two pages.

8 Simulation Laws for Data Refinement

Program refinement investigates the stepwise transformation of abstract specifications to executable code. Data refinement is a variant that considers the transformation of *abstract* data types (ADTs) such as sets into *concrete* ADTs such as lists, stacks or queues. Abstract ADTs are observed through the effects of their operations on states, and operations are usually modelled as binary relations. Two further operations model the initialisation and finalisation of ADTs with respect to a global state space. By definition, an abstract ADT is refined by a concrete ADT if the relation induced by all execution sequences of abstract operations between an abstract initialisation and an abstract finalisation is contained in the relation induced by the corresponding concrete sequences. To replace this by a local criterion, abstraction relations are introduced that relate inputs and outputs of operations at the abstract and the concrete level. de Roever and Engelhardt's book contains further information [9]. Program refinement often requires inequational reasoning. Therefore we used isotonicity of multiplication and transitivity as additional hypothesis in our experiments. Named by de Roever and Engelhardt according to the shape of the corresponding diagrams, U-simulations, L-simulations and their converses can be considered. Formally, let x, y and z be elements of some relation algebra. Then

– x *U-simulates* y with respect to z ($x \subseteq_U^z y$) if $\breve{z}; x; z \leq y$,
– x *L-simulates* y with respect to z ($x \subseteq_L^z y$) if $\breve{z}; x \leq y; \breve{z}$,
– x *$\breve{U}$-simulates* y with respect to z ($x \subseteq_{\breve{U}}^z y$) if $x \leq z; y; \breve{z}$,
– x *$\breve{L}$-simulates* y with respect to z ($x \subseteq_{\breve{L}}^z y$) if $x; z \leq z; y$.

In all these definitions, z is the *abstraction relation* and $\subseteq$ the *simulation relation*. We now consider compositionality properties of simulations.

Theorem 8.1 **([9]).** *Let z be a simulation relation.*

(i) $x_1 \subseteq_U^z y_1$ *and* $x_2 \subseteq_U^z y_2$ *imply* $x_1; x_2 \subseteq_U^z y_1; y_2$ *if z is total.*
(ii) $x_1 \subseteq_{\breve{U}}^z y_1$ *and* $x_2 \subseteq_{\breve{U}}^z y_2$ *imply* $x_1; x_2 \subseteq_{\breve{U}}^z y_1; y_2$ *if z is a function.*
(iii) $x_1 \subseteq_L^z y_1$ *and* $x_2 \subseteq_L^z y_2$ *imply* $x_1; x_2 \subseteq_L^z y_1; y_2$*, and similarly for* $\breve{L}$.

Proof. Prover9 needed less than 3 *s* for each individual claim. □

Theorem 8.2 **([9]).** $x \subseteq_L^{z_1} y$ *and* $x \subseteq_L^{z_2} y$ *imply* $x \subseteq_L^{z_1; z_2} y$*, and similarly for the other simulations.*

Proof. Prover9 needed less than 1 *s* for the each implication. □

Also the implications between different simulations could be automated.

Theorem 8.3 **([9]).**

(i) *If the simulation relation is a function then $\breve{U}$-simulation implies L-, $\breve{L}$- and U-simulation, and U-simulation is implied by L- and $\breve{L}$-simulation.*
(ii) *If the simulation relation is total then U-simulation implies L-, $\breve{L}$ and $\breve{U}$-simulation, and $\breve{U}$-simulation is implied by L- and $\breve{L}$-simulation.*

Proof. Prover9 presented proofs for both claims after less than 1 *s*. □

We now prove simulation laws for iterations of relations. In relation algebras over complete Boolean algebras, the finite iteration of an element x is defined through the reflexive transitive closure

$$x^* = x^0 + x^1 + x^2 + \ldots = \sup(x^i : i \geq 0) \ ,$$

where $x^0 = 1$ and $x^{i+1} = x; x^i$. This definition is rather useless for ATP systems. We therefore use the well known unfold and induction laws

$$1 + x; x^* \leq x^* \ , \qquad 1 + x^*; x \leq x^* \ ,$$
$$z + x; y \leq y \Rightarrow x^*; z \leq y \ , \qquad z + y; x \leq y \Rightarrow z; x^* \leq y \ .$$

The proof that these laws hold in relation algebras over complete Boolean algebras requires a simple induction. Based on this first-order encoding, a series of laws could easily be verified automatically. Some example experiments are listed in Table 11; more can be found at our website.

Table 11. Relational Iteration Laws

#	Theorem	t[s]
(62) (63)	$x^*; x^* = x^*$ $(x^*)^* = x^*$	2.55
(64)	$x; y \leq y \Rightarrow x^*; y \leq y$	0.02
(65)	$z; x \leq y; z \Rightarrow z; x^* \leq y^*; z$	2.20
(66)	$(x; y)^*; x = x; (y; x)^*$	2.05

To speed up proofs we used the join splitting law $x + y \leq z \Leftrightarrow x \leq z \wedge y \leq z$ as an additional hypothesis in the proofs of (65) and (66).

The properties from Table 11 yield useful hypotheses for automating de Roever and Engelhardt's soundness proofs of simulations for data refinement. Soundness of simulations means that the existence of a simulation between the particular operations of ADTs implies that there is a data refinement. In the sequel, we restrict our attention to L-simulations.

Theorem 8.4 ([9]). *L-simulations are sound for data refinement.*

Proof. Let i^a, x_i^a and f^a denote the initialisation, the operations and the finalisation at the abstract level; let i^c, x_i^c and f^c denote the corresponding relations at the concrete level. We can assume that $i^c \leq i^a; \breve{z}$, $x^c \subseteq_L^z x^a$ and $\breve{z}; f^c \leq f^a$ holds for arbitrary atomic operations x^a and x^c. We must show that $i^c; s^c; f^c \leq i^a; s^a; f^a$ holds for arbitrary sequences s^a and s^c of operations.

The proof uses structural induction over s^a. We first prove that $s^c \subseteq_L^z s^a$ holds for some s^c. The entire induction can, of course, not be treated by ATPs, but the particular base cases and induction steps can.

We consider the empty operation 0 and the skip operation 1 as base cases. Prover9 showed in 9.95 s that $0 \subseteq_L^z 0$ and $1 \subseteq_L^z 1$. The case of atomic operations holds by assumption. For the induction step we consider abstract operations of the form $s_1^a; s_2^a$, $s_1^a + s_2^a$ and $(s^a)^*$.
(i) Let $s_1^c \subseteq_L^z s_1^a$ and $s_2^c \subseteq_L^z s_2^a$. But $s_1^c; s_2^c \subseteq_L^z s_1^a; s_2^a$ has already been shown in Theorem 8.1.

(ii) Let $s_1^c \subseteq_L^z s_1^a$ and $s_2^c \subseteq_L^z s_2^a$. Using a distributivity law as additional hypothesis, Prover9 needed $1.78\,s$ to show that $s_1^c + s_2^c \subseteq_L^z s_1^a + s_2^a$.
(iii) Let $s^c \subseteq_L^z s^a$. For the automated proof we used the unfold and induction laws of reflexive transitive closure and added Equation (65) as an additional hypothesis. Then Prover9 could show that $(s^c)^* \subseteq_L^z (s^a)^*$ in less than $1\,s$.

For the final step, assume that $i^c \le i^a; \breve{z}$, $\breve{z}; f^c \le f^a$ and $s^c \subseteq_L^z s^a$, which has just been shown. Prover9 then showed in $0.53\,s$ that $i^c; s^c; f^c \le i^a; s^a; f^a$. $\square$

Automated proofs for the remaining simulations are also straightforward. Interestingly, de Rover and Engelhardt do not mention that U must be total and $\breve{U}$ must be a function in these proofs; Mace4 immediately found counterexamples to Part (iii) of these proofs without these assumptions. Here, we also used Equation (66) instead of (65) as additional hypothesis.

L- and $\breve{L}$-simulations are also complete for data refinement, but proofs in the literature are pointwise (cf. [9]) and additional effort would be required to extract a purely relation-algebraic proof. We leave this for future work.

9 Outlook

Relations can not only model abstract data types. They also provide standard semantics for imperative and functional programs. In the imperative case, it is very natural to model the input/output behaviour of a program as a binary relation between states encoded as vectors. The standard weakest liberal precondition semantics for partial correctness and the weakest precondition semantics for total correctness can be defined in the setting of relation algebra [5,20]. The wlp-operator for a program x and a state p can be defined as $\mathsf{wlp}(x,p) = \overline{x; \overline{p}}$. The wp-operators with respect to a program x, a state p and a vector $\tau(x)$ denoting the guaranteed termination of x can be defined as $\mathsf{wp}(x,p) = \mathsf{wlp}(x,p) \cdot \tau(x)$. Standard laws of the $\mathsf{w(l)p}$-calculi such as $\mathsf{wlp}(x+y,p) = \mathsf{wlp}(x,p) \cdot \mathsf{wlp}(y,p)$, $\mathsf{wp}(x+y,p) = \mathsf{wp}(x,p) \cdot \mathsf{wp}(y,p)$, $\mathsf{wlp}(x,p \cdot q) = \mathsf{wlp}(x,p) \cdot \mathsf{wlp}(x,q)$ or $\mathsf{wp}(x,p \cdot q) = \mathsf{wp}(x,p) \cdot \mathsf{wp}(x,q)$ could then be automatically derived without any difficulties.

Our examples suggest that a combination of relation algebras with ATP systems could contribute to make formal methods more automatic and user-friendly. All current verification tools for formal methods like B or Z are highly interactive. Although the translation of system specifications into relational semantics is rather simple and can yield very concise expressions, the manipulation of relational expressions in verification tasks is usually cumbersome for non-experts. Encapsulating the calculus as far as possible by using ATP behind the scenes could improve this situation. Practical verification tasks often require the integration of algebraic techniques into a wider context: Most induction proofs require higher-order reasoning, but the base case and the induction step can often be discharged algebraically. Other applications might require pointwise reasoning with concrete functions and relations and with assignments. But pointwise properties can often be abstracted into bridge-lemmas and proofs then confined to the abstract algebraic layer. In this sense we envisage the integration of relation algebras into ATP systems as a novel light-weight formal method that should be extended by higher-order techniques and combined with decision procedures.

10 Conclusion

We automatically verified more than hundred theorems of relation algebras with Prover9 and Mace4. Many of these proofs were considered worth publishing by eminent mathematicians some decades ago and most students would probably still find them difficult. Our experiments suggest that the automation of relation algebras with off-the-shelf theorem provers is feasible. This presents an interesting alternative to higher-order, special-purpose, translational and finitist approaches. The statements proved form a basic library that can safely be used and extended. Our results pave the way for interesting applications in relational software development methods and automated deduction with modal logics. A larger case study, in which the experiments of this paper have been replayed with other ATP systems [8], confirms our results.

We envisage three main directions for further work. First, to be more useful in formal methods, ways of combining the abstract pointfree level with the concrete level of data need to be developed. Second, an integration of ordered chaining techniques [3] into modern ATP systems would certainly make relational reasoning, which is predominantly inequational, more efficiently. Third, a combination with more powerful hypothesis learning techniques seems indispensable for tackling more complex applications and larger specifications. The obvious impact on formal verification technology makes these tasks certainly worth pursuing.

References

1. `http://www.dcs.shef.ac.uk/~georg/ka`
2. Abrial, J.-R.: The B-Book. Cambridge University Press, Cambridge (1996)
3. Bachmair, L., Ganzinger, H.: Ordered chaining calculi for first-order theories of transitive relations. J. ACM 45(6), 1007–1049 (1998)
4. Berghammer, R., Neumann, F.: An OBDD-based computer algebra system for relations. In: Ganzha, V.G., Mayr, E.W., Vorozhtsov, E.V. (eds.) CASC 2005. LNCS, vol. 3718, pp. 40–51. Springer, Heidelberg (2005)
5. Berghammer, R., Zierer, H.: Relational algebraic semantics of deterministic and nondeterministic programs. Theoretical Computer Science 43, 123–147 (1986)
6. Bird, R., de Moor, O.: Algebra of Programming. Prentice-Hall, Englewood Cliffs (1996)
7. Blackburn, P., de Rijke, M., Venema, Y.: Modal Logic. Cambridge University Press, Cambridge (2001)
8. Dang, H.-H., Höfner, P.: First-order theorem prover evaluation w.r.t. relation- and Kleene algebra. In: R. Berghammer, B. Möller, and G. Struth, editors, RelMiCS10/AKA5 - PhD Programme, Technical Report 2008-04, University of Augsburg, pp. 48–52 (2008)

9. de Roever, W.-P., Engelhardt, K.: Data Refinement: Model-Oriented Proof Methods and their Comparison. Cambridge University Press, Cambridge (2001)
10. Formisano, A., Omodeo, E.G., Orłowska, E.: A Prolog tool for relational translations of modal logics: A front-end for relational proof systems. In: Beckert, B. (ed.) TABLEAUX 2005. LNCS (LNAI), vol. 3702, pp. 1–11. Springer, Heidelberg (2005)
11. Henkin, L., Monk, D.J., Tarski, A.: Cylindric Algebras, Part I. North-Holland, Amsterdam (1971)
12. Huntington, E.V.: Boolean algebra. A correction. Trans. AMS 35, 557–558 (1933)
13. Huntington, E.V.: New sets of independent postulates for the algebra of logic. Trans. AMS 35, 274–304 (1933)
14. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, Cambridge (2006)
15. Jónsson, B., Tarski, A.: Boolean algebras with operators I. American J. Mathematics 74, 891–939 (1951)
16. Kahl, W.: Calculational relation-algebraic proofs in Isabelle/Isar. In: Berghammer, R., Möller, B., Struth, G. (eds.) RelMiCS 2003. LNCS, vol. 3051, pp. 179–190. Springer, Heidelberg (2004)
17. MacCaull, W., Orlowska, E.: Correspondence results for relational proof systems with application to the Lambek calculus. Studia Logica 71(3), 389–414 (2002)
18. Maddux, R.: A sequent calculus for relation algebras. Annals of Pure and Applied Logic 25, 73–101 (1983)
19. Maddux, R.: Relation Algebras. Elsevier, Amsterdam (2006)
20. Maddux, R.D.: Relation-algebraic semantics. Theoretical Computer Science 160(1&2), 1–85 (1996)
21. McCune, W.: Prover9 and Mace4, http://www.cs.unm.edu/~mccune/prover9
22. McCune, W.: Solution of the Robbins problem. J. Automated Reasoning 19(3), 263–276 (1997)
23. Schmidt, G., Ströhlein, T.: Relations and Graphs: Discrete Mathematics for Computer Scientists. Springer, Heidelberg (1993)
24. Sinz, C.: System description: ARA — An automated theorem prover for relation algebras. In: McAllester, D. (ed.) CADE 2000. LNCS, vol. 1831, pp. 177–182. Springer, Heidelberg (2000)
25. Spivey, J.M.: Understanding Z. Cambridge University Press, Cambridge (1988)
26. Sutcliffe, G., Suttner, C.: The TPTP problem library: CNF release v1.2.1. J. Automated Reasoning 21(2), 177–203 (1998)
27. Sutcliffe, G., Suttner, C.: Evaluating general purpose automated theorem proving systems. Artificial Intelligence 131(1–2), 39–54 (2001)
28. Tarski, A.: On the calculus of relations. Journal of Symbolic Logic 6(3), 73–89 (1941)
29. Tarski, A., Givant, S.R.: A Formalization of Set Theory Without Variables. American Mathematical Society (1987)
30. von Oheimb, D., Gritzner, T.F.: Rall: Machine-supported proofs for relation algebra. In: McCune, W. (ed.) CADE 1997. LNCS, vol. 1249, pp. 380–394. Springer, Heidelberg (1997)
31. Weidenbach, C., Schmidt, R.A., Hillenbrand, T., Rusev, R., Topic, D.: System description: SPASS version 3.0. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 514–520. Springer, Heidelberg (2007)

A Relation Algebra in TPTP-style

```
%---- Boolean algebra (Huntington)
fof(join_commutativity,axiom,(
    ! [X0,X1] : join(X0,X1) = join(X1,X0) )).

fof(associativity,axiom,(
    ! [X0,X1,X2] : join(X0,join(X1,X2)) = join(join(X0,X1),X2) )).

fof(Huntington,axiom,(
    ! [X0,X1] : X0 = join(complement(join(complement(X0),complement(X1))),
                         complement(join(complement(X0),X1))) )).

fof(meet_definiton,axiom,(
    ! [X0,X1] : meet(X0,X1) = complement(join(complement(X0),complement(X1))) )).

%---- Sequential Composition
fof(composition_associativity,axiom,(
    ! [X0,X1,X2] : composition(X0,composition(X1,X2)) =
                   composition(composition(X0,X1),X2) )).

fof(composition_identity,axiom,(
    ! [X0] : composition(X0,one) = X0 )).

fof(composition_distributivity,axiom,(
    ! [X0,X1,X2] : composition(join(X0,X1),X2) =
                   join(composition(X0,X2),composition(X1,X2)) )).

%---- Converse
fof(converse_idempotence,axiom,(
    ! [X0] : converse(converse(X0)) = X0 )).

fof(converse_additivity,axiom,(
    ! [X0,X1] : converse(join(X0,X1)) = join(converse(X0),converse(X1)) )).

fof(converse_multiplicativity,axiom,(
    ! [X0,X1] : converse(composition(X0,X1)) =
                composition(converse(X1),converse(X0)) )).

fof(converse_cancellativity,axiom,(
    ! [X0,X1] : join(composition(converse(X0),complement(composition(X0,X1))),
                complement(X1)) = complement(X1) )).
```

Towards SMT Model Checking of Array-Based Systems

Silvio Ghilardi[1], Enrica Nicolini[2], Silvio Ranise[1,2], and Daniele Zucchelli[1]

[1] Dipartimento di Informatica, Università degli Studi di Milano, Italia
[2] LORIA & INRIA-Lorraine, Nancy, France

Abstract. We introduce the notion of array-based system as a suitable abstraction of infinite state systems such as broadcast protocols or sorting programs. By using a class of quantified-first order formulae to symbolically represent array-based systems, we propose methods to check safety (invariance) and liveness (recurrence) properties on top of Satisfiability Modulo Theories solvers. We find hypotheses under which the verification procedures for such properties can be fully mechanized.

1 Introduction

Model checking of infinite-state systems manipulating arrays – e.g., broadcast protocols, lossy channel systems, or sorting programs – is a hot topic in verification. The key problem is to verify the correctness of such systems regardless of the number of elements (processes, data, or integers) stored in the array, called *uniform verification problem*. In this paper, we propose *array-based systems* as a suitable abstraction of broadcast protocols, lossy channel systems, or, more in general, programs manipulating arrays (Section 3). The notion of array-based system is parametric with respect to a theory of indexes (which, for parametrized systems, specifies the topology of processes) and a theory of elements (which, again for parametrized systems, specifies the data manipulated by the system). Then (Section 3.1), we show how states and transitions of a large class of array-based systems can be symbolically represented by a class of *quantified* first-order formulae whose satisfiability problem is decidable under reasonable hypotheses on the theories of indexes and elements (Section 4). We also sketch how to extend the *lazy Satisfiability Modulo Theories* (SMT) techniques [16] by a suitable *instantiation strategy* to handle universally quantified variables (over indexes) so as to implement the satisfiability procedure for the class of quantified formulae under consideration (Figure 2). The capability to handle a (limited form) of universal quantification is crucial to reduce entailment between formulae representing states to satisfiability of a formula in the class of quantified formulae previously defined. This observation together with closure under pre-image computation of the kind of formulae representing unsafe states allows us to implement a *backward* reachability procedure (see, e.g., [12], or also [15] for a declarative approach) for checking safety properties of array-based systems (Section 5). In general, the procedure may not terminate;

but, under some additional assumptions on the theory of elements, we are able to prove termination by revisiting the notion of configuration (see, e.g., [1]) in first-order model-theory (Section 5.2). Finally (Section 6), we show the flexibility of our approach by studying the problem of checking the class of liveness properties called *recurrences* [13]. We devise deductive methods to check such properties based on the synthesis of so-called progress conditions at quantifier free level and we show how to fully mechanize our technique under the same hypotheses for the termination of the backward reachability procedure.

For space reasons, we did not include proofs; for the same reason, although our framework covers all examples discussed in [2,3] (and more), only one running example is discussed. For both proofs and examples, the reader is referred to the Technical Report [10].

2 Formal Preliminaries

We assume the usual first-order syntactic notions of signature, term, formula, quantifier free formula, and so on; equality is always included in our signatures. If $\underline{x}$ is a finite set of variables and Σ is a signature, by a $\Sigma(\underline{x})$-term, -formula, etc. we mean a term, formula, etc. in which at most the $\underline{x}$ occur free (notations like $t(\underline{x})$, $\varphi(\underline{x})$ emphasize the fact that the term t or the formula φ in in fact a $\Sigma(\underline{x})$-term or a $\Sigma(\underline{x})$-formula, respectively). The notions of interpretation, satisfiability, validity, and logical consequence are also the standard ones; when we speak about satisfiability (resp. validity) of a formula containing free variables, we mean satisfiability of its existential (resp. universal) closure. If $\mathcal{M} = (M, \mathcal{I})$ is a Σ-structure, a *Σ-substructure* of $\mathcal{M}$ is a Σ-structure having as domain a subset of M which is closed under the operations of Σ (in a Σ-substructure, moreover, the interpretation of the symbols of Σ is given by restriction). The Σ-structure *generated by a subset* X of M (which is not assumed now to be closed under the Σ-operations) is the smallest Σ-substructure of $\mathcal{M}$ whose domain contains X and, if this Σ-substructure coincides with the whole $\mathcal{M}$, we say that X *generates* $\mathcal{M}$. A *Σ-embedding* (or, simply, an embedding) between two Σ-structures $\mathcal{M} = (M, \mathcal{I})$ and $\mathcal{N} = (N, \mathcal{J})$ is any mapping $\mu : M \longrightarrow N$ among the corresponding support sets which is an isomorphism between $\mathcal{M}$ and the Σ-substructure of $\mathcal{N}$ whose underlying domain is the image of μ (thus, in particular, for μ to be an embedding, the image of μ must be closed under the Σ-operations). A class $\mathcal{C}$ of structures is *closed under substructures* iff whenever $\mathcal{M} \in \mathcal{C}$ and $\mathcal{N}$ is (isomorphic to) a substructure of $\mathcal{M}$, then $\mathcal{N} \in \mathcal{C}$.

Contrary to previous papers of ours, we prefer to have here a more liberal notion of a theory, so we identify a *theory* T with a pair $(\Sigma, \mathcal{C})$, where Σ is a signature and $\mathcal{C}$ is a class of Σ-structures (the structures in $\mathcal{C}$ are called the *models* of T). The notion of *T-satisfiability* of φ means the satisfiability of φ in a Σ-structure from $\mathcal{C}$; similarly, *T-validity* of a sentence φ (noted $T \models \varphi$) means the truth of φ in all $\mathcal{M} \in \mathcal{C}$. The *Satisfiability Modulo Theory T*, SMT(T), problem amounts to establish the T-satisfiability of an arbitrary first-order formula (hence possibly containing quantifiers), w.r.t. some background theory T.

A *theory solver* for the theory T (T-solver) is any procedure capable of establishing whether any given finite conjunction of literals is T-satisfiable or not. The so-called lazy approach to solve SMT(T) problems for quantifier-free formulae consists of integrating a Boolean enumerator (usually based on a refinement of the DPLL algorithm) with a T-solver (see [14,16] for an overview). Hence, by assuming the existence of a T-solver, it is always possible to build a (lazy SMT) solver capable of checking the T-satisfiability of arbitrary Boolean combinations of atoms in T (or, equivalently, quantifier-free formulae). For efficiency, a T-solver is required to provide a more refined interface, such as returning a subset of a T-unsatisfiable input set of literals which is still T-unsatisfiable, called conflict set (again see [16] for details).

We say that T admits *quantifier elimination* iff for every formula $\varphi(\underline{x})$ one can compute a quantifier-free formula $\varphi'(\underline{x})$ which is T-equivalent to it (i.e. such that $T \models \forall \underline{x}(\varphi(\underline{x}) \leftrightarrow \varphi'(\underline{x}))$). Linear Arithmetics, Real Arithmetics, acyclic lists, and enumerated datatype theories (see below) admit elimination of quantifiers.

A theory $T = (\Sigma, \mathcal{C})$ is said to be *locally finite* iff Σ is finite and, for every finite set of variables $\underline{x}$, there are finitely many $\Sigma(\underline{x})$-terms $t_1, \dots, t_{k_{\underline{x}}}$ such that for every further $\Sigma(\underline{x})$-term u, we have that $T \models u = t_i$ (for some $i \in \{1, \dots, k_{\underline{x}}\}$). The terms $t_1, \dots, t_{k_{\underline{x}}}$ are called $\Sigma(\underline{x})$-representative terms; if they are effectively computable from $\underline{x}$ (and t_i is computable from u), then T is said to be *effectively locally finite* (in the following, when we say 'locally finite', we in fact always mean 'effectively locally finite'). If Σ is finite and does not contain any function symbol (i.e. Σ is a purely relational signature), then any Σ-theory is effectively locally finite; other effectively locally finite theories are Boolean algebras and Linear Arithmetic modulo a fixed integer.

Let Σ be a finite signature; an *enumerated datatype theory* in Σ is a theory whose class of models contains only a single finite Σ-structure $\mathcal{M} = (M, \mathcal{I})$; we require $\mathcal{M}$ to have the additional property that for every $m \in M$ there is a constant $c \in \Sigma$ such that $c^{\mathcal{I}} = m$. It is easy to see that an enumerated datatype theory admits quantifier elimination (since $\exists x\, \varphi(x)$ is T-equivalent to $\varphi(c_1) \vee \cdots \vee \varphi(c_n)$, where $c_1, ..., c_n$ are interpreted as the finitely many elements of the enumerated datatype) and is effectively locally finite.

In the following, it is more natural to adopt a many-sorted language. All notions introduced above (such as term, formula, structure, and satisfiability) can be easily adapted to many-sorted logic, see e.g. Chapter XX of [8].

3 Array-Based Systems and Their Symbolic Representation

We develop a framework to state and solve model checking problems for safety and liveness of a particular class of infinite state systems by using deductive (more precisely, SMT) techniques. We focus on the class of systems whose state can be described by a finite collections of arrays, which we call *array-based* systems. As an example, consider parameterized systems, i.e. systems consisting of an arbitrary number of identical finite-state processes organized in a linear array: a state a of such a parameterized system can be seen as a state of an array-based system (in the formal sense of Definitions 3.2 and 3.3 below), where the indexes of the domain of the function assigned to the state variable a are the identifiers of the processes and the elements of a are the data describing one of the finitely many states of each process.

Array-based Systems. To develop our formal model, we introduce three theories. We use two mono-sorted theories $T_I = (\Sigma_I, \mathcal{C}_I)$ (of indexes) and $T_E = (\Sigma_E, \mathcal{C}_E)$ (of elements), whose only sort symbol are called `INDEX` and `ELEM`, respectively. T_I specifies the 'topology' of the processes, e.g., in the case of parameterized systems informally discussed above, Σ_I contains only the equality symbol '=' and $\mathcal{C}_I$ is the class of all finite sets. Other interesting examples of T_I can be obtained by taking Σ_I to contain a binary predicate symbol R (besides =) and $\mathcal{C}_I$ to be the class of structures where R is interpreted as a total order, a graph, a forest, etc. For parameterized systems with finite-state processes, T_E can be the theory of an enumerated datatype. For the larger class of parameterized systems (e.g., the one considered in [2,3]) admitting integer (or real) variables local to each process, T_E can be the theory whose class of models consists of a single structure (like real numbers under addition and/or ordering). Notice that in concrete applications, T_E has a single model (e.g. an enumerate datatype, or the structure of real/natural numbers under suitable operations and relations), whereas T_I has many models (e.g. it has as models all sets, all finite sets, all graphs, all finite graphs, etc.). The technical hypotheses we shall need in Sections 4 and 5 on T_I and T_E are fixed in the following:

Definition 3.1. *An* index theory $T_I = (\Sigma_I, \mathcal{C}_I)$ *is a mono-sorted theory (let us call* `INDEX` *its sort) which is locally finite, closed under substructures and whose quantifier free fragment is decidable for T_I-satisfiability. An* element theory $T_E = (\Sigma_E, \mathcal{C}_E)$ *is a mono-sorted theory (let us call* `ELEM` *its sort) which admits quantifier elimination and whose quantifier free fragment is decidable for T_E-satisfiability.*

One may wonder how restrictive are the assumptions of the above definition: it turns out that they are very light (for instance, they do not rule out any of the examples considered in [2,3]). In fact, quantifier elimination holds for common datatype theories (integers, reals, enumerated datatypes, etc.) and the hypotheses on index theory are satisfied by most process 'topologies'.[1]

The third theory $A_I^E = (\Sigma, \mathcal{C})$ we need is obtained by combining an index theory T_I and an element theory T_E as follows. First, A_I^E has three sort symbols: `INDEX`, `ELEM`, and `ARRAY`; the signature Σ contains all the symbols in the disjoint union $\Sigma_I \cup \Sigma_E$ and a further binary function symbol *apply* of sort `ARRAY` $\times$ `INDEX` $\longrightarrow$ `ELEM`. (In the following, we abbreviate $apply(a, i)$ with $a[i]$, for a term of sort `ARRAY` and i term of sort `INDEX`.) Second, a three-sorted structure $\mathcal{M} = (\mathtt{INDEX}^{\mathcal{M}}, \mathtt{ELEM}^{\mathcal{M}}, \mathtt{ARRAY}^{\mathcal{M}}, \mathcal{I})$ is in the class $\mathcal{C}$ iff $\mathtt{ARRAY}^{\mathcal{M}}$ is the set

[1] They typically fail when processes are arranged in a ring (e.g. in the 'dining philosophers' example): rings require a unary function symbol to be formalized and the bijectivity constraint to be imposed is insufficient to make the theory locally finite.

of (total) functions from $\texttt{INDEX}^{\mathcal{M}}$ to $\texttt{ELEM}^{\mathcal{M}}$, the function symbol *apply* is interpreted as function application (i.e. as the standard reading operation for arrays), and $(\texttt{INDEX}^{\mathcal{M}}, \mathcal{I}_{|\Sigma_I})$, $(\texttt{ELEM}^{\mathcal{M}}, \mathcal{I}_{|\Sigma_E})$ are models of T_I and T_E, respectively – here $\mathcal{I}_{|\Sigma_I}, \mathcal{I}_{|\Sigma_E}$ are the restriction of $\mathcal{I}$ to the symbols of Σ_I, Σ_E. (In the following, we use the notations $\mathcal{M}_I$ and $\mathcal{M}_E$ for $(\texttt{INDEX}^{\mathcal{M}}, \mathcal{I}_{|\Sigma_I})$ and $(\texttt{ELEM}^{\mathcal{M}}, \mathcal{I}_{|\Sigma_E})$, respectively.) If the model $\mathcal{M}$ of A_I^E is such that $\texttt{INDEX}^{\mathcal{M}}$ is a finite set, then $\mathcal{M}$ is called a *finite index model.*

For the remaining part of the paper, **we fix an index theory** $T_I = (\Sigma_I, \mathcal{C}_I)$ **and an element theory,** $T_E = (\Sigma_E, \mathcal{C}_E)$ (we also let $A_I^E = (\Sigma, \mathcal{C})$ be the corresponding combined theory).

Once the theories constraining indexes and elements are fixed, we need the notions of state, initial state and state transition to complete our picture. In a symbolic setting, these are provided by the following definitions:

Definition 3.2. *An* array-based (transition) system (for (T_I, T_E)) *is a triple* $\mathcal{S} = (\underline{a}, I, \tau)$ *where:*

- $\underline{a}$ *is a tuple of variables of sort* $\texttt{ARRAY}$ *(these are the* state variables*);*
- $I(\underline{a})$ *is a* $\Sigma(\underline{a})$*-formula (this is the* initial state formula*);*
- $\tau(\underline{a}, \underline{a}')$ *is a* $\Sigma(\underline{a}, \underline{a}')$*-formula – here* $\underline{a}'$ *is a renamed copy of the tuple* $\underline{a}$ *(this is the* transition *formula).*

Definition 3.3. *Let* $\mathcal{S} = (\underline{a}, I, \tau)$ *be an array-based transition system. Given a model* $\mathcal{M}$ *of the combined theory* A_I^E*, an* $\mathcal{M}$-state *(or, simply, a state) of* $\mathcal{S}$ *is an assignment mapping the state variables* $\underline{a}$ *to total functions* $\underline{s}$ *from the domain of* $\mathcal{M}_I$ *to the domain of* $\mathcal{M}_E$*. A* run *of the array-based system is a (possibly infinite) sequence* $\underline{s}_0, \underline{s}_1, \ldots$ *of states such that*[2] $\mathcal{M} \models I(\underline{s}_0)$ *and* $\mathcal{M} \models \tau(\underline{s}_k, \underline{s}_{k+1})$ *for* $k \geq 0$*.*

For simplicity, below, we assume that the tuple of array state variables $\underline{a}$ is a single variable a: all definitions and results of the paper can be easily generalized to the case of finitely many array variables and to the case of multi-sorted T_I, T_E (see [10] for a discussion on these topics).

3.1 Symbolic Representation of States and Transitions

The next step is to identify suitable syntactic restrictions for the formulae I, τ appearing in the definition of an array-based system. Preliminarily, we introduce some notational conventions that alleviate the burden of writing and understanding the various formulae for states and transitions: $d, e, \ldots$ range over variables of sort $\texttt{ELEM}$, $a, b \ldots$ over variables of sort $\texttt{ARRAY}$, $i, j, k, \ldots$ over variables of sort $\texttt{INDEX}$, and $\alpha, \beta, \ldots$ over variables of either sort $\texttt{ELEM}$ or sort $\texttt{INDEX}$. An underlined variable name abbreviates a tuple of variables of unspecified (but finite) length; by subscripting with an integer an underlined variable name, we indicate the corresponding component of the tuple (e.g., $\underline{i}$ may abbreviate $i_1, \ldots, i_n$

[2] Notations like $\mathcal{M} \models I(\underline{s}_0)$ means that the formula $I(\underline{a})$ is true in $\mathcal{M}$ under the assignment mapping the $\underline{a}$'s to the $\underline{s}_0$'s.

and $\underline{i}_k$ indicates i_k, for $1 \leq k \leq n$). We also use $a[\underline{i}]$ to abbreviate the tuple of terms $a[i_1], \ldots, a[i_n]$. If $\underline{j}$ and $\underline{i}$ are tuples with the same length n, then $\underline{i} = \underline{j}$ abbreviates $\bigwedge_{k=1}^{n} (\underline{i}_k = \underline{j}_k)$. Possibly sub/super-scripted expressions of the forms $\phi(\underline{\alpha}), \psi(\underline{\alpha}), \ldots$ (with sub-/super-scripts) always denote **quantifier-free** $(\Sigma_I \cup \Sigma_E)$**-formulae in which at most the variables** $\underline{\alpha}$ **occur** (notice in particular that no array variable and no apply constructor $a[i]$ can occur here). Also, $\phi(\underline{\alpha}, \underline{t}/\underline{\beta})$ (or simply $\phi(\underline{\alpha}, \underline{t})$) abbreviates the substitution of the terms $\underline{t}$ for the variables $\underline{\beta}$ (now apply constructors may appear in t). Thus, for instance, when we write $\phi(\underline{i}, a[\underline{i}])$, we mean the formula obtained by the replacements $\underline{e} \mapsto a[\underline{i}]$ in the quantifier free formula $\phi(\underline{i}, \underline{e})$ (the latter contains at most the element variables $\underline{e}$ and at most the index variables $\underline{i}$).

We are now ready to introduce suitably restricted classes of formulae for representing states and transitions of array-based systems.

States. An $\exists^I$**-formula** is a formula of the form $\exists \underline{i}\, \phi(\underline{i}, a[\underline{i}])$: such a formula may be used to model sets of *unsafe* states of parameterized systems, such as violations of mutual exclusion:

$$\exists i\, \exists j\, (i \neq j \wedge a[i] = \texttt{use} \wedge a[j] = \texttt{use}), \tag{1}$$

where `use` is a constant symbol of sort `ELEM` in an enumerated datatype theory. A $\forall^I$**-formula** is a formula of the form $\forall \underline{i}\, \phi(\underline{i}, a[\underline{i}])$: this is logically equivalent to the negation of an $\exists^I$-formula and may be used to model *initial* sets of states of parameterized systems, such as "all processes are in a given state":

$$\forall i\, (a[i] = \texttt{idle}),$$

where `idle` is a constant symbol of sort `ELEM` in an enumerated datatype theory.

Transitions. The intuition underlying the class of formulae representing transitions of array-based systems can be found by analyzing the structure of transitions of parameterized systems. In such systems, a transition has typically two components. The *local* component (see the formula ϕ_L in the Definition 3.4 below) specifies the transitions of a given fixed number of processes; the *global* component (see the formula ϕ_G in the Definition 3.4 below) specifies the transitions done by all the other processes in the array as a reaction to those taken by the processes involved in the local component. We are lead to the following:

Definition 3.4. *Consider formulae* $\phi_L(\underline{i}, \underline{d}, \underline{d}')$ *and* $\phi_G(\underline{i}, \underline{d}, \underline{d}', j, e, e')$*, where* ϕ_G *satisfies the following* seriality *requirement:*

$$A_I^E \models \forall \underline{i}\, \forall \underline{d}\, \forall \underline{d}'\, \forall j\, \forall e\, \exists e'\, \phi_G(\underline{i}, \underline{d}, \underline{d}', j, e, e'). \tag{2}$$

The **T-formula** *with local component* ϕ_L *and global component* ϕ_G *is the formula*

$$\exists \underline{i}\, (\phi_L(\underline{i}, a[\underline{i}], a'[\underline{i}]) \wedge \mathit{Update}_G(\underline{i}, a, a')), \tag{3}$$

where we used the explicit definition (i.e. the abbreviation)

$$Update_G(\underline{i}, a, a') :\Leftrightarrow \forall j\ \phi_G(\underline{i}, a[\underline{i}], a'[\underline{i}], j, a[j], a'[j]). \tag{4}$$

Recall that, according to our conventions, the local component ϕ_L and the global component ϕ_G are quantifier-free $(\Sigma_I \cup \Sigma_E)$-formulae. In $\phi_G(\underline{i}, \underline{d}, \underline{d}', j, e, e')$ the variables $\underline{i}, \underline{d}, \underline{d}'$ are called *side* parameters (these variables are instantiated in (3)-(4) by $\underline{i}, a[\underline{i}], a'[\underline{i}]$, respectively) and the variables j, e, e' are called *proper* parameters (these variables are instantiated in (3)-(4) by $j, a[j], a'[j]$, respectively). The intuition underlying the formula $Update_G(\underline{i}, a, a')$ defined by (4) is that the array a' is obtained as *a global update of the array a according to* ϕ_G. In fact, in (4), the proper parameters of ϕ_G are instantiated as $j, a[j], a'[j]$, i.e. as the index to be updated, the old, and the new content of that index. Notice also that this updating is largely non-deterministic, because a T-formula like (3) does not univocally characterize the system evolution: this is revealed by the fact that ϕ_G is not a definable function, and by the fact that the side parameters $\underline{d}'$ that may occur in ϕ_G are instantiated as the updated values $a'[\underline{i}]$ (this circularity makes sense only if we view the T-formula (3) as expressing a *constraint* – not necessarily an assignment – for the updated array a').

In the remaining part of the paper, **we fix an array-based system $\mathcal{S} = (a, I, \tau)$, in which the initial formula I is a $\forall^I$-formula and the transition formula τ is a *disjunction of* T-formulae**.

Example 3.5. We present here the formalization into our framework of the *Simplified Bakery Algorithm* that can be found for instance in [3]. We take as T_I the pure equality theory in the signature $\Sigma_I = \{=\}$; to introduce T_E, we analyze which kind of local data we need. The data e appearing in an array of processes are records consisting of the following two fields: (i) the field $e.s$ represents the status ($e.s \in \{\texttt{idle}, \texttt{wait}, \texttt{use}, \texttt{crash}\}$); (ii) the field $e.t$ represents the ticket (tickets range in the real interval $[0, \infty)$, seen as a linear order with first element 0). Thus we need a two-sorted T_E and two array variables in the language (or, a single array variable and a three-sorted T_E, comprising a sort for the cartesian product): as pointed out above, such multi-sorted extensions of our framework are indeed straightforward. Models for T_E are the obvious ones: one sort is interpreted as an enumerated datatype and the other sort is interpreted as the positive real domain. The initial formula I is $\forall i\ (a[i].s = \texttt{idle})$. The transition $\tau(a, a')$ is the disjunction $\tau_1(a, a') \vee \tau_2(a, a') \vee \tau_3(a, a')$, where the T-formulae τ_n $(n \leq 3)$ have the standard format from Definition 3.4 – namely $\exists i\ (\phi^n_L(i, a[i], a'[i]) \wedge Update^n_G(i, a, a'))$ – and the local and the global components ϕ^n_L, ϕ^n_G are specified in Figure 1 below. When formalizing the component transitions τ_1, τ_2, we use the *approximation* trick (see [2,3]): transitions τ_1, τ_2, in order to fire, would require a universally quantified guard which cannot be expressed in our format. We circumvent the problem by imposing that all the processes that are counterexamples to the guard go into a ‘crash’ state: this trick augments the possible runs of the system by introducing ‘spurious runs’, however if a safety property holds for the augmented ‘approximate’ system, then it also holds for the original system (the same is true for the recurrence properties of Section 6). $\dashv$

Local components. The local components ϕ_L^n, for $n \leq 3$, are

- $\phi_L^1 \quad :\equiv \quad a[i].s = \texttt{idle} \wedge a'[i].s = \texttt{wait} \wedge a'[i].t > 0$ (the selected process goes from the state `idle` to `wait` and takes a nonzero ticket);
- $\phi_L^2 \quad :\equiv \quad a[i].s = \texttt{wait} \wedge a'[i] = \langle \texttt{use}, a[i].t \rangle$ (the selected process goes from the state `wait` to `use` and keeps the same ticket);
- $\phi_L^3 \quad :\equiv \quad a[i].s = \texttt{use} \wedge a'[i] = \langle \texttt{idle}, 0 \rangle$ (the selected process goes from the state `use` to `idle` and the ticket is reset to zero).

Global components. The global components $\phi_G^n(i, a[i], a'[i], j, a[j], a'[j])$, for $n \leq 3$, make case distinction as follows (the first case delegates the update to the local component):

- $\phi_G^1 \quad :\equiv \quad (j = i \wedge a'[j] = a'[i]) \vee (j \neq i \wedge a[j].t < a[i].t \wedge a'[j] = a[j]) \vee (j \neq i \wedge a[j].t \geq a[i].t \wedge a'[j] = \langle \texttt{crash}, a[j].t \rangle)$ (if the selected process i goes from `idle` to `wait`, then any other process j keeps the same state/ticket, unless it has a ticket bigger or equal to the ticket of i – in which case j crashes);
- $\phi_G^2 \quad :\equiv \quad (j = i \wedge a'[j] = a'[i]) \vee (j \neq i \wedge (a[j].t > a[i].t \vee a[j].t = 0) \wedge a'[j] = a[j]) \vee (j \neq i \wedge \neg(a[j].t > a[i].t \vee a[j].t = 0) \wedge a'[j] = \langle \texttt{crash}, a[j].t \rangle)$ (if the selected process i goes from `wait` to `use`, then any other process j keeps the same state/ticket, unless it has a nonzero ticket smaller than the ticket of i – in which case j crashes);
- $\phi_G^3 \quad :\equiv \quad (j = i \wedge a'[j] = a'[i]) \vee (j \neq i \wedge a'[j] = a[j])$ (if the selected process i goes from `use` to `idle`, then any other process keeps the same state/ticket).

Fig. 1. The Simplified Bakery Transition

4 Symbolic Representation and SMT Solving

To make effective use of our framework, we need to be able to check whether certain requirements are met by a given array-based system. In other words, we need a powerful reasoning engine: it turns out that A_I^E-satisfiability of $\exists^{A,I}\forall^I$-sentences introduced below is just what we need.[3]

Theorem 4.1. *The A_I^E-satisfiability of $\exists^{A,I}\forall^I$-sentences, i.e. of sentences of the kind*

$$\exists a_1 \cdots \exists a_n \; \exists \underline{i} \; \forall \underline{j} \; \psi(\underline{i}, \underline{j}, a_1[\underline{i}], \ldots, a_n[\underline{i}], a_1[\underline{j}], \ldots, a_n[\underline{j}]), \tag{5}$$

is decidable.

We leave to [10] the detailed proof of the above theorem, here we focus on a discussion oriented to the employment of SMT-solvers in the decision procedure: Figure 2 depicts an SMT-based decision procedure for $\exists^{A,I}\forall^I$-sentences (in the simplified case in which only one variable of sort `ARRAY` occurs).

[3] Decidability of A_I^E-satisfiability of $\exists^{A,I}\forall^I$-sentences plays in this paper a role similar to the Σ_1^0-decidability result employed in [4].

```
function $A_I^E$-check($\exists a\ \exists \underline{i}\ \forall \underline{j}\ \psi(\underline{i}, \underline{j}, a[\underline{i}], a[\underline{j}])$)
    $\underline{t} \longleftarrow$ compute-reps$_{T_I}(\underline{i})$; $\phi \longleftarrow \underline{t} = \underline{l}$; $Atoms \longleftarrow$ IE($\underline{l}$)
    for each substitution $\sigma$ mapping the $\underline{j}$ into the $\underline{t}$'s do
        $\phi' \longleftarrow$ purify($\psi(\underline{i}, \underline{j}\sigma, a[\underline{i}], a[\underline{j}\sigma])$); $\phi \longleftarrow \phi \wedge \phi'$; $Atoms \longleftarrow Atoms \cup$ atoms($\phi'$);
    end for each
    while Bool($\phi$) = sat do
        $\beta_I \wedge \beta_E \wedge \eta_I \longleftarrow$ pick-assignment($Atoms, \phi$)
        $(\rho_I, \pi_I) \longleftarrow T_I$-solver($\beta_I \wedge \eta_I$)
        $(\rho_E, \pi_E) \longleftarrow T_E$-solver($\beta_E \wedge \bigwedge_{\underline{l}_{s_1} = \underline{l}_{s_2} \in \eta_I} (\underline{e}_{s_1} = \underline{e}_{s_2})$)
        if $\rho_I$ = sat and $\rho_E$ = sat then return sat
        if $\rho_I$ = unsat then $\phi \longleftarrow \phi \wedge \neg\pi_I$
        if $\rho_E$ = unsat then $\phi \longleftarrow \phi \wedge \neg\pi_E$
    end while
    return unsat
end
```

Fig. 2. The SMT-based decision procedure for $\exists^{A,I}\forall^I$-sentences

The set $\underline{t}$ of representative terms over $\underline{i}$ is computed by compute-reps$_{T_I}$ (to simplify the matter, we can freely assume $\underline{t} \supseteq \underline{i}$). This function is guaranteed to exist since T_I is effectively locally finite (for example, when Σ_I contains no function symbol, compute-reps$_{T_I}$ is the identity function). In order to purify formulae, we shall need below fresh index variables $\underline{l}$ abstracting the $\underline{t}$ and fresh element variables $\underline{e}$ abstracting the terms $a[\underline{l}]$: the conjunction of the 'defining equations' $\underline{t} = \underline{l}$ is stored as the formula ϕ, whereas the further defining equations $a[\underline{l}] = \underline{e}$ (not to be sent to the Boolean solver) will be taken into account inside the second loop body.

Then, the first loop is entered where we instantiate in all possible ways the universally quantified index variables $\underline{j}$ with the representative terms in the set $\underline{t}$. For every such substitution σ, the formula $\psi(\underline{i}, \underline{j}\sigma, a[\underline{i}], a[\underline{j}\sigma])$ is purified by replacing the terms $a[\underline{t}]$ with the element variables $\underline{e}$; after such purification, the resulting formula is added as a new conjunct to ϕ. Notice that this ϕ is a quantifier-free formula whose atoms are pure, i.e. they are either Σ_I- or Σ_E-atoms. The function IE($\underline{l}$) returns the set of all possible equalities among the index variables $\underline{l}$. Such equalities are added to the set of atoms occurring in ϕ as the T_I- and T_E-solvers need to take into account an equivalence relation over the index variables $\underline{l}$ so as to synchronize.

We can now enter the second (and main) loop: the Boolean solver, by invoking the interface function pick-assignment inside the loop, generates an assignment over the set of atoms occurring in ϕ and IE($\underline{l}$). The loop is exited when ϕ is checked unsatisfiable (test of the while). The T_I-solver checks for unsatisfiability the set β_I of Σ_I-literals in the Boolean assignment and the literals of the possible partition η_I on $\underline{l}$. Then, only the equalities $\bigwedge\{\underline{e}_{s_1} = \underline{e}_{s_2} \mid (\underline{l}_{s_1} = \underline{l}_{s_2}) \in \eta_I\}$ (and not also inequalities) among the purification variables $\underline{e}$ induced by the partition

η_I on $\underline{l}$ are passed to the T_E-solver together with the set β_E of Σ_E-literals in the Boolean assignment (this is to take into account the congruence of $a[\underline{l}] = \underline{e}$). If both solvers detect satisfiability, then the loop is exited and A_I^E-check also returns satisfiability. Otherwise (i.e. if at least one of the solvers returns unsat), the negation of the computed conflict set (namely, π_I or π_E) is conjoined to ϕ so as to refine the Boolean abstraction and the loop is resumed. If in the second loop, satisfiability is never detected for all considered Boolean assignments, then A_I^E-check returns unsatisfiability. The second loop can be seen as a refinement of the Delayed Theory Combination technique [5].

The critical point of the above procedure is the fact that the first loop may produce a very large ϕ: in fact, even in the most favourable case in which compute-reps$_{T_I}$ is the identity function, the purified problem passed to the solvers of the second loop has exponential size (this is consequently the dominating cost of the whole procedure, see [10] for complexity details). To improve performances, an *incremental* approach is desirable: in the incremental approach, the second loop may be entered before all the possible substitutions have been examined. If the second loop returns unsat, one can exit the whole algorithm and only in case it returns sat further substitutions (producing new conjuncts for ϕ) are taken into consideration. Since when A_I^E-check is called in conclusive steps of our model checking algorithms (for fixpoint, safety, or progress checks), the expected answer is unsat, this incremental strategy may be considerably convenient. Other promising suggestions for reducing the number of instantiations (in the case of particular index and elements theories) come from recent decision procedures for fragments of theories of arrays, see [6,11].

5 Safety Model Checking

Checking safety properties means checking that a set of 'bad states' (e.g., of states violating the mutual exclusion property) cannot be reached by a system. This can be formalized in our settings as follows:

Definition 5.1. *Let $K(a)$ be an $\exists^I$-formula (whose role is that of symbolically representing the set of bad states). The* safety model checking problem *for K is the problem of deciding whether there is an $n \geq 0$ such that the formula*

$$I(a_0) \wedge \tau(a_0, a_1) \wedge \cdots \wedge \tau(a_{n-1}, a_n) \wedge K(a_n) \tag{6}$$

is A_I^E-satisfiable. If such an n does not exist, K is safe*; otherwise, it is* unsafe.

Notice that the A_I^E-satisfiability of (6) means precisely the existence of a finite run leading from a state in I to a state in K.

5.1 Backward Reachability

If a bound for n in the safety model checking is known a priori, then safety can be decided by checking the A_I^E-satisfiability of suitable instances of (6) (this is possible because each formula (6) is equivalent to a $\exists^{A,I}\forall^{I}$-formula). Unfortunately, this is rarely the case and we must design a more refined method. In the following, we adapt algorithms that incrementally maintain a representation of the set of reachable states in an array-based system.

Definition 5.2. *Let $K(a)$ be an $\exists^I$-formula. An $\mathcal{M}$-state s_0 is* backward reachable from K in n steps *iff there exists a sequence of $\mathcal{M}$-states $s_1, \ldots, s_n$ such that*

$$\mathcal{M} \models \tau(s_0, s_1) \wedge \cdots \wedge \tau(s_{n-1}, s_n) \wedge K(s_n).$$

We say that s_0 is backward reachable from K *iff it is backward reachable from K in n steps for some $n \geq 0$.*

Before designing our backward reachability algorithm, we must give a symbolic representation of the set of backward reachable states. Preliminarily, notice that if $K(a)$ is an $\exists^I$-formula, the set of states which are backward reachable in one step from K can be represented as follows:

$$Pre(\tau, K) := \exists a' \, (\tau(a, a') \wedge K(a')). \tag{7}$$

Although $Pre(\tau, K)$ is not an $\exists^I$-formula anymore, we are capable of finding an equivalent one in the class:

Proposition 5.3. *Let $K(a)$ be an $\exists^I$-formula; then $Pre(\tau, K)$ is A_I^E-equivalent to an (effectively computable) $\exists^I$-formula $K'(a)$.*

The proof of this Proposition can be obtained by a tedious syntactic computation while using (2) together with the fact that T_E admits quantifier elimination.

Abstractly, the set of states that can be backward reachable from K can be recursively characterized as follows: (a) $Pre^0(\tau, K) := K$ and (b) $Pre^{n+1}(\tau, K) := Pre(\tau, Pre^n(\tau, K))$. The formula $BR^n(\tau, K) := \bigvee_{s=0}^{n} Pre^s(\tau, K)$ (having just one free variable of type `ARRAY`) symbolically represents the states that can be backward reached from K in n steps. The sequence of $BR^n(\tau, K)$ allows us to rephrase the safety model checking problem (Definition 5.1) using symbolic representations by saying that K *is safe iff the formulae $I \wedge BR^n(\tau, K)$ are all not A_I^E-satisfiable.* This still requires infinitely many tests, *unless there is an n such that the formula $\neg(BR^{n+1}(\tau, K) \rightarrow BR^n(\tau, K))$ is A_I^E-unsatisfiable.*[4] The key observation now is that both $I \wedge BR^n(\tau, K)$ and $\neg(BR^{n+1}(\tau, K) \rightarrow BR^{n+1}(\tau, K))$ can be easily transformed into $\exists^{A,I}\forall^I$-formulae so that their A_I^E-satisfiability is decidable and checked by the procedure A_I^E-check of Figure 2.

This discussion suggests the adaptation of a standard backward reachability algorithm (see, e.g., [12]) depicted in Figure 3, where Pre takes a formula, it computes Pre as defined in (7), and then applies the required syntactic manipulations to transform the resulting formula into an equivalent one according to Proposition 5.3.

[4] Notice that the A_I^E-unsatisfiability of $\neg(BR^n(\tau, K) \rightarrow BR^{n+1}(\tau, K))$ is obvious by definition. Hence, the A_I^E-unsatisfiability of $\neg(BR^{n+1}(\tau, K) \rightarrow BR^n(\tau, K))$ implies that $A_I^E \models BR^{n+1}(\tau, K) \leftrightarrow BR^n(\tau, K)$.

```
function BReach($K$)
    $i \longleftarrow 0$; $BR^0(\tau, K) \longleftarrow K$; $K^0 \longleftarrow K$
    if $A_I^E$-check($BR^0(\tau, K) \wedge I$) = sat then return unsafe
    repeat
        $K^{i+1} \longleftarrow \mathsf{Pre}(\tau, K^i)$
        $BR^{i+1}(\tau, K) \longleftarrow BR^i(\tau, K) \vee K^{i+1}$
        if $A_I^E$-check($BR^{i+1}(\tau, K) \wedge I$) = sat then return unsafe
        else $i \longleftarrow i + 1$
    until $A_I^E$-check($\neg(BR^{i+1}(\tau, K) \rightarrow BR^i(\tau, K))$) = unsat
    return safe
end
```

Fig. 3. Backward Reachability for Array-based Systems

Theorem 5.4. *Let $K(a)$ be an $\exists^I$-formula; then the function* BReach *in Figure 3 semi-decides the safety model checking problem for K.*

Example 5.5. In the Simplified Bakery example of Figure 1 (with the formula K to be tested for safety given like in (1)), the algorithm of Figure 3 stops after four loops and certifies safety of K. $\dashv$

5.2 Termination of Backward Reachability

Indeed, the main problem with the semi-algorithm of Figure 3 is termination; in fact, it may not terminate when the system is safe. In the literature, termination of infinite state model checking is often obtained by defining a well-quasi-ordering (wqo – see, e.g., [1]) on the configurations of the system. Here, we adapt this approach to our setting, *by defining model-theoretically a notion of configuration and then introducing a suitable ordering on configurations*: once this is done, it will be immediate to prove that termination follows whenever the ordering among configurations is a wqo.

An A_I^E-*configuration* (or, briefly, a configuration) is an $\mathcal{M}$-state in a finite index model $\mathcal{M}$ of A_I^E: a configuration is denoted as $(s, \mathcal{M})$, or simply as s, leaving $\mathcal{M}$ implicit. Moreover, we associate a Σ_I-structure s_I and a Σ_E-structure s_E with an A_I^E-configuration $(s, \mathcal{M})$ as follows: the Σ_I-structure s_I is simply the finite structure $\mathcal{M}_I$, whereas s_E is the Σ_E-substructure of $\mathcal{M}_E$ generated by the image of s (in other words, if $\texttt{INDEX}^{\mathcal{M}} = \{c_1, \dots, c_k\}$, then s_E is generated by $\{s(c_1), \dots, s(c_k)\}$).

We remind few definitions about preorders. A preorder $(P, \leq)$ is a set endowed with a reflexive and transitive relation; an *upset* of such a preorder is a subset $U \subseteq P$ such that ($p \in U$ and $p \leq q$ imply $q \in U$). An upset U is *finitely generated* iff it is a finite union of cones, where a *cone* is an upset of the form $\uparrow p = \{q \in P \mid p \leq q\}$ for some $p \in P$. A preorder $(P, \leq)$ is a *well-quasi-ordering* (wqo) iff every upset of P is finitely generated (this is equivalent to the standard definition, see [10]). We define now a preorder among our configurations:

Definition 5.6. *Let s, s' be configurations; $s' \leq s$ holds iff there are a Σ_I-embedding $\mu : s'_I \longrightarrow s_I$ and a Σ_E-embedding $\nu : s'_E \longrightarrow s_E$ such that the set-theoretical compositions of μ with s and of s' with ν are equal.*

Let $K(a)$ be an $\exists^I$-formula: we denote by $[\![K]\!]$ the set of A_I^E-configurations satisfying K; in symbols, $[\![K]\!] := \{(\mathcal{M}, s) \mid \mathcal{M} \models K(s)\}$.

Proposition 5.7. *For every $\exists^I$-formula $K(a)$, the set $[\![K]\!]$ is upward closed; also, for every $\exists^I$-formulae K_1, K_2, we have $[\![K_1]\!] \subseteq [\![K_2]\!]$ iff $A_I^E \models K_1 \to K_2$.*

The set of configurations $\mathcal{B}(\tau, K)$ which are backward reachable from a given $\exists^I$-formula K is thus an upset, being the union of infinitely many upsets; however, even in case the latter are finitely generated, $\mathcal{B}(\tau, K)$ needs not be so. Under the hypothesis of local finiteness of T_E, this is precisely what characterizes termination of backward reachability search:

Theorem 5.8. *Assume that T_E is locally finite; let K be an $\exists^I$-formula. If K is safe, then* BReach *in Figure 3 terminates iff $\mathcal{B}(\tau, K)$ is a finitely generated upset.*[5] *As a consequence,* BReach *always terminates when the preorder on A_I^E-configurations is a wqo.*

The termination result of Theorem 5.8 covers many well-known special cases, like broadcast protocols, some versions of the bakery algorithm, lossy channel systems (notice that for the latter, there doesn't exist a primitive recursive complexity lower bound); to apply Theorem 5.8, one needs classical facts like Dikson's Lemma, Higman's Lemma, Kruskal's Theorem, etc. (see again [10] for details).

6 Progress Formulae for Recurrence Properties

Liveness problems are difficult and even more so for infinite state systems. In the following, we consider a special kind of liveness properties, which falls into the class of recurrence properties in the classification introduced in [13]. Our method for dealing with such properties consists in synthesizing progress functions definable at quantifier free level.

A recurrence property is a property which is *infinitely often* true in every infinite run of a system; in other words, to check that R is a recurrence property it is sufficient to show that it cannot happen that there is an infinite run $s_0, s_1, \ldots, s_i, \ldots$ such that R is true at most in the states from a finite prefix $s_0, \ldots, s_m$. This can be formalized in our framework as follows.

Definition 6.1. *Suppose that $R(a)$ is a $\forall^I$-formula; let us use the abbreviations $K(a)$ for $\neg R(a)$ and $\tau_K(a, a')$ for $\tau(a, a') \wedge K(a) \wedge K(a')$.*[6] *We say that R is a* recurrence property *iff for every m the infinite set of formulae*

[5] If K is unsafe, we already know that BReach terminates because it detects unsafety (cf. Theorem 5.4).

[6] Notice that τ_K is easily seen to be equivalent to a disjunction of T-formulae, like the original τ.

$$I(b_1), \tau(b_1, b_2), \ldots, \tau(b_m, a_0), \tau_K(a_0, a_1), \tau_K(a_1, a_2), \ldots \qquad (8)$$

is not satisfiable in a finite index model of A_I^E.

The finite prefix $I(b_1), \tau(b_1, b_2), \ldots, \tau(b_m, a_0)$ in (8) above ensures that the states assigned to $a_0, a_1, \ldots$ are (forward) reachable from an initial state, whereas the infinite suffix $\tau_K(a_0, a_1)$, $\tau_K(a_1, a_2), \ldots$ expresses the fact that property R is never attained. Observe that, whereas it can be shown that the A_I^E-satisfiability of (6) for safety problems (cf. Definition 5.1) is equivalent to its satisfiability in a finite index model of A_I^E, this is not the case for (8). This imposes the above explicit restriction to finite index models since, otherwise, recurrence might unnaturally fail in many concrete situations.

Example 6.2. In the Simplified Bakery example of Figure 1, we take K to be

$$\exists i\ (i = d \wedge a[i] = \texttt{wait})$$

(here d is a fresh constant of type Σ_I). Then the A_I^E-satisfiability of (8) implies that the protocol cannot guarantee absence of starvation. $\dashv$

Let R be a recurrence property; we say that R has *polynomial complexity* iff there exists a polynomial $f(n)$ such that for every m and for $k > f(n)$ the formulae

$$I(b_1) \wedge \tau(b_1, b_2) \wedge \cdots \wedge \tau(b_m, a_0) \wedge \tau_K(a_0, a_1) \wedge \cdots \wedge \tau_K(a_{k-1}, a_k) \qquad (9)$$

are all unsatisfiable in the models $\mathcal{M}$ of A_I^E such that the cardinality of the support of $\mathcal{M}_I$ is at most n (in other words, $f(n)$ gives an upper bound for the waiting time needed to reach R by a system consisting of less than n processes).

Definition 6.3. *Let $R(a)$ be a $\forall^I$-formula and let K and τ_K be as in Definition 6.1. A formula $\psi(\underline{t}(\underline{j}), a[\underline{t}(\underline{j})])$ is an* index invariant *iff there exists a safe $\exists^I$-formula H such that*

$$A_I^E \models \forall a_0\ \forall a_1\ \forall \underline{j}\ (\neg H(a_0) \wedge \tau_K(a_0, a_1) \to (\psi(\underline{t}, a_0[\underline{t}]) \to \psi(\underline{t}, a_1[\underline{t}]))). \qquad (10)$$

A progress condition *for R is a finite set of index invariant formulae*

$$\psi_1(\underline{t}(\underline{j}), a[\underline{t}(\underline{j})]), \ldots, \psi_c(\underline{t}(\underline{j}), a[\underline{t}(\underline{j})])$$

for which there exists a safe $\exists^I$-formula H such that

$$A_I^E \models \forall a_0\, \forall a_1\, \big(\neg H(a_0) \wedge \tau_K(a_0, a_1) \to \exists \underline{j} \bigvee_{k=1}^{c} (\neg \psi_k(\underline{t}, a_0[\underline{t}]) \wedge \psi_k(\underline{t}, a_1[\underline{t}]))\big). \quad (11)$$

Suppose there is a progress condition as above for R; if $\mathcal{M}$ is a model of A_I^E such that the support of $\mathcal{M}_I$ has cardinality at most n, then the formula (9) cannot be satisfiable in $\mathcal{M}$ for $k > c \cdot n^\ell$ (here ℓ is the length of the tuple $\underline{j}$). This argument shows the following:

Proposition 6.4. *If there is a progress condition for R, then R is a recurrence property with polynomial complexity.*

Example 6.5. In the Simplified Bakery example of Example 6.2, the following four index invariant formulae:

$$a[j].s = \texttt{crash} \qquad a[j].t = 0 \lor a[j].t > a[d].t$$
$$a[j].t > a[d].t \qquad \neg(a[j].t < a[d].t \land a[j].s = \texttt{wait})$$

constitute a progress condition, thus guaranteeing that each non faulty process d can get (in polynomial time) the resource every time it asks for it. The algorithm of Figure 3 is needed for certifying safety of some formulae (like $\exists i\,(a[i].s = \texttt{use} \land a[i].t = 0)$) to be used as the H of Definition 6.3. $\dashv$

Let us now discuss how to mechanize the application of Proposition 6.4. The validity tests (10) and (11) can be reduced to unsatisfiability tests covered by Theorem 4.1; however, the search space for progress conditions looks to be unbounded for various reasons (the number of the ψ's of Definition 6.3, the length of the string $\underline{j}$, the 'oracle' safe H's, etc.). Nevertheless, the proof of the following unexpected result shows (in the appropriate hypotheses) how to find progress conditions whenever they exist:

Theorem 6.6. *Suppose that T_E is locally finite and that safety of $\exists^I$-formulae can be effectively checked. Then the existence of a progress condition for a given $\forall^I$-formula R is decidable.*

7 Related Work and Conclusions

A popular approach to uniform verification (e.g., [2,3,4]) consists of defining a finitary representation of the (infinite) set of states and then to explore the state space by using such a suitable data structure. Although such a data structure may contain declarative information (e.g., constraints over some algebraic structure or first-order formulae), a substantial part is non-declarative. Typically, the part of the state handled non-declaratively is that of indexes (which, for parametrized systems, specify the topology), thereby forcing to re-design the verification procedures whenever their specification is changed (for the case of parametrized systems, whenever the topology of the systems changes). Since our framework is fully declarative, we avoid this problem altogether.

There has been some attempts to use a purely logical techniques to check both safety and liveness properties of infinite state systems (e.g., [7,9]). The hard part of these approaches is to guess auxiliary assertions (either invariants, for safety, or ranking functions, for liveness). Indeed, finding such auxiliary assertions is not easy and automation requires techniques adapted to specific domains (e.g., integers). In this paper, we have shown how our approach does not require auxiliary invariants for safety and proved that the search for certain progress conditions can be fully mechanized (Theorem 6.6).

In order to test the viability of our approach, we have built a prototype implementation of the backward reachability algorithm of Figure 3. The function A_I^E-check has been implemented by using a naive instantiation algorithm and a state-of-the-art SMT solver. We have also implemented the generation of the proof obligations (10) and (11) for recognizing progress conditions. This allowed us to automatically verify the examples discussed in this paper and in [10]. We are currently planning to build a robust implementation of our techniques.

References

1. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.-K.: General decidability theorems for infinite state systems. In: Proc. of LICS 1996, pp. 313–321 (1996)
2. Abdulla, P.A., Delzanno, G., Ben Henda, N., Rezine, A.: Regular model checking without transducers. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 721–736. Springer, Heidelberg (2007)
3. Abdulla, P.A., Delzanno, G., Rezine, A.: Parameterized verification of infinite-state processes with global conditions. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 145–157. Springer, Heidelberg (2007)
4. Bouajjani, A., Habermehl, P., Jurski, Y., Sighireanu, M.: Rewriting systems with data. In: Csuhaj-Varjú, E., Ésik, Z. (eds.) FCT 2007. LNCS, vol. 4639, pp. 1–22. Springer, Heidelberg (2007)
5. Bozzano, M., Bruttomesso, R., Cimatti, A., Junttila, T.A., Ranise, S., van Rossum, P., Sebastiani, R.: Efficient theory combination via boolean search. Information and Computation 204(10), 1493–1525 (2006)
6. 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 (2005)
7. 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 (LNAI), vol. 2392, pp. 438–455. Springer, Heidelberg (2002)
8. Enderton, H.B.: A Mathematical Introduction to Logic. Academic Press, London (1972)
9. Fang, Y., Piterman, N., Pnueli, A., Zuck, L.D.: Liveness with invisible ranking. Software Tools for Technology 8(3), 261–279 (2006)
10. Ghilardi, S., Nicolini, E., Ranise, S., Zucchelli, D.: Towards SMT model checking of array-based systems. Technical Report RI318-08, Università degli Studi di Milano (2008), `http://homes.dsi.unimi.it/~zucchell/publications/techreport/GhiNiRaZu-RI318-08.pdf`
11. Ihlemann, C., Jacobs, S., Sofronie-Stokkermans, V.: On local reasoning in verification. In: Proc. of TACAS 2008. LNCS, vol. 4963, pp. 265–281. Springer, Heidelberg (2008)
12. Kesten, Y., Maler, O., Marcus, M., Pnueli, A., Shahar, E.: Symbolic model checking with rich assertional languages. Theoretical Computer Science 256(1-2), 93–112 (2001)
13. Manna, Z., Pnueli, A.: A hierarchy of temporal properties. In: Proc. of PODC 1990, pp. 377–410. ACM Press, New York (1990)
14. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL. Journal of the ACM 53(6), 937–977 (2006)
15. 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)
16. Sebastiani, R.: Lazy satisfiability modulo theories. Journal on Satisfiability, Boolean Modeling and Computation 3, 141–224 (2007)

Preservation of Proof Obligations from Java to the Java Virtual Machine[⋆]

Gilles Barthe[1], Benjamin Grégoire[2], and Mariela Pavlova[3]

[1] IMDEA Software, Madrid, Spain
[2] INRIA Sophia-Antipolis Méditerranée, France
[3] Trusted Labs, France

Abstract. While program verification environments typically target source programs, there is an increasing need to provide strong guarantees for executable programs.

We establish that it is possible to reuse the proof that a source Java program meets its specification to show that the corresponding JVM program, obtained by non-optimizing compilation, meets the same specification. More concretely, we show that verification condition generators for Java and JVM programs generate the same set of proof obligations, when applied to a program p and its compilation $[\![p]\!]$ respectively.

Preservation of proof obligations extends the applicability of Proof Carrying Code, by allowing certificate generation to rely on existing verification technology.

1 Introduction

As program verification environments (PVEs) are reaching maturity, they are being increasingly used for validating large programs that are typically written in high-level programming languages such as Java and C. While reasoning about source programs provides developers with a direct feedback on the code they write, there are compelling reasons to seek achieving similar guarantees for compiled code. As emphasized by Reps and co-workers, see e.g. [1], a first reason is that source code is not what you execute; in particular, code that has been verified at source level may be invalidated by a compiler (even if the compiler preserves the input/output behavior of programs). A second reason is that program verification finds many of its applications in the field of mobile code, where the code consumer may not have access to the source code, and may not trust the code developer nor the network through which the code is transferred to him: as a result, code consumers need evidence about the compiled code which they receive and execute.

One possible approach to solve the mismatch is to develop verification environments for compiled programs, but the approach has major drawbacks, especially in the context of interactive verification: one looses the benefit of reasoning on a structured language, and the verification effort is needlessly duplicated, as each program must be verified once per target language and compiler. A better solution is to transfer evidence from

[⋆] Most of the work was performed while the first and third authors were working at INRIA.The work was partially supported by the MOBIUS project.

source code to compiled programs, so that verification can be performed as usual with existing tools, and that code is only proved once.

In this paper, we consider the problem of transferring evidence from Java programs to JVM programs (JVM stands for Java Virtual Machine, which provides part of the runtime environment for executing Java programs). We focus on verification methods based on verification condition generations, as such methods prevail in PVEs. In this setting, we show that it is equivalent to verify a source program or its compilation, in the sense that they generate the *same* proof obligations, and thus one can reuse exactly the same proofs for Java programs and for their compilation as JVM programs (in the idealized setting considered in this paper, *same* is to be understood as syntactic equality; Section 6 discusses how the proof obligations are only almost the same in practice). This strong equivalence between verification conditions at source and bytecode levels, which we call preservation of proof obligations (PPO), has significant implications in the context of Proof Carrying Code (PCC) [12]: in effect, PPO supports the following extension of the Java slogan: write *and prove* once, *verify and* run everywhere. By letting the code consumer prove interactively the correctness of its application at source level, and sending the resulting certificate to the code consumer, PPO extends the applicability of PCC to complex properties.

2 Related Work

Barthe, Rezk and Saabas [5] showed preservation of proof obligations in the setting of a simple imperative language. As in the present paper, they consider verification condition generators that operate directly on the syntax of the program, and focus on a non-optimizing compiler. Burdy and Pavlova [6] subsequently considered preservation of proof obligations in the context of Java, and showed on an example that proof obligations were almost preserved; in particular, they identified similar issues to the ones reported in Section 6. Pavlova [16] later formalized the relationship between verification at Java and JVM levels for a language that extends the fragment considered here with arrays and constructors, and for a logic that uses frame conditions. Our work is directly inspired from [16], but abstracts from some specificities of Java in order to increase readability.

Verification condition generation can be seen as a specific strategy for constructing proofs in Hoare logics, and some authors have studied the problem of preserving provability of Hoare triples by compilation, providing effective algorithms to transform derivations (one can also show that provability is preserved by compilation by invoking the soundness and relative completeness of the logic and the preservation of semantics by compilation, see the discussion on certified compilation below, but it is of limited usefulness for PCC). Saabas and Uustalu [18] develop a compositional proof systems for assembly programs with "phrase structure", and transforms proofs of source programs into proofs of compiled programs, for an extended compiler that generates automatically the "phrase structure" of programs. Bannwart and Müller [2] also define a Hoare logic for sequential fragments of Java source code and bytecode, and illustrate how derivations of correctness can be mapped from source programs to bytecode programs obtained by non-optimizing compilation. Müller and Nordio [11] subsequently showed the correctness of a proof transforming procedure for a fragment with abrupt termination. More recently, Nordio, Müller and Meyer [15] define a proof transforming procedure from Eiffel to Microsoft's Common Intermediate Language; the procedure has been formalized using Isabelle [14]. Although preservation of provability of Hoare triples by compilation is closely related to preservation of proof obligations, we believe that our results provide a stronger correspondence for verification architectures based on verification condition generation.

Hoare logics can be seen as instances of abstract interpretations [7], and preservation of proof obligations has a natural generalization in the setting of abstract interpretation. Rival [17] showed how the results of abstract interpretation could be transferred from source to compiled programs. However, Logozzo and Fähndrich [10] pointed out that provability by automated means such as abstract interpretation is not always preserved in a straightforward way because some static analyses are less precise on compiled code than on source code.

To the exception of [17], the above works consider non-optimizing compilers. While preservation of proof obligations fails in presence of program optimizations, one can still generate a proof of correctness of the compiled program from a proof of correctness of the source program, provided one strengthens the invariants with the results of the analyses that justify the transformation. Barthe *et al* [3] show how to define certificate translations for many common program optimizations; this work was further systematized in [4], using a mild extension of the framework of abstract interpretation to provide sufficient conditions for preserving provability by compilation. In a series of papers, see e.g. [19], Saabas and Uustalu show similar results, in the context of Hoare logics.

Besides the above mentioned works that share similar motivations to ours, there are many works that explore the interaction between compilation and verification. Certifying compilation [13] focuses on the automatic generation of certificates for simple safety properties; obviously, certifying shares many commonalities with our work, but favors automation at the price of limiting the scope of the policies that can be considered. Yet another alternative is provided by certified compilers [9], which are implemented and verified in a proof assistant; since compiler verification consists in proving semantics-preservation, one can use certified compilers to transform evidence of source programs into evidence of compiled programs. However, this approach currently suffers from two major limitations: the resulting certificates are huge, and can only refer to input/output properties of programs.

3 Setting

This section introduces the fragment of Java and of the Java Virtual Machine considered, and define the non-optimizing compiler for which preservation of proof obligations shall be established. In addition to omitting some features (such as local variables, constructors, interfaces, or arrays), we abstract away from specificities that are irrelevant from the point of view of preservation of proof obligations (e.g. that the class `Object` is at the root of the hierarchy of classes). Furthermore, we depart from the class file format of the JVM in our treatment of names, by using named variables instead of

operations	$\mathsf{op} ::= + \mid - \mid \times \mid /$		
comparisons	$\mathsf{cmp} ::= < \mid \leq \mid = \mid \neq \mid \geq \mid >$		
expressions	$e ::= x \mid n \mid \mathsf{null} \mid e \; \mathsf{op} \; e \mid \mathsf{this} \mid e.\mathsf{m}(e) \mid e.\mathsf{f} \mid \mathsf{new}\; C$		
tests	$t ::= e \; \mathsf{cmp} \; e$		
statements	$s ::= x := e$		assignment
	$\mid$	skip	skip
	$\mid$	$\mathsf{if}(t)\{s\}\{s\}$	conditional
	$\mid$	$\mathsf{while}(t)\{s\}$	loop
	$\mid$	$e.\mathsf{f} := e$	field assignment
	$\mid$	$s;s$	sequence
	$\mid$	$\mathsf{return}\; e$	return value
	$\mid$	$\mathsf{try}\{s\}\;\mathsf{catch}\;(E)\;\{s\}$	try/catch
	$\mid$	$\mathsf{throw}\; e$	throw

Fig. 1. Java language

relying on a constant pool and an index for local variables; see Section 6. Due to space constraints, we do not present the intended semantics of Java and JVM programs.

Java and JVM programs. The structure of programs is common to Java and the JVM. A (Java or JVM) program is specified by a set $\mathcal{M}$ of method names, a set $\mathcal{C}$ of class names, partially ordered by a subclass relation $\preceq$, by a set $\mathcal{F}$ of field identifiers, and by its method declarations and class declarations. The set $\mathcal{T}$ of types is defined as $\mathcal{C} \cup \{\mathsf{int}\}$, where int is the type of integers (see Section 6 for a discussion on basic types); we use $\mathcal{T}_\perp$ to denote the union of $\mathcal{T}$ with void, and use it to define the signature of methods. We also assume given a subset $\mathcal{E}$ of $\mathcal{C}$ that represent the exception classes.

Definition 1. *A program p is defined by a pair* $(\mathbb{M}, \mathbb{C})$ *where* $\mathbb{C} = (\mathcal{C} \times \mathcal{F}) \rightharpoonup \mathcal{T}$ *is a class declaration map, and* $\mathbb{M} = (\mathbb{M}_{\mathsf{type}}, \mathbb{M}_{\mathsf{code}})$ *is a method declaration table, i.e.:*

- $\mathbb{M}_{\mathsf{type}}$ *assigns to every method a method signature of the form* $\sigma \to \tau$, *where* $\sigma \in \mathcal{T}$ *and* $\tau \in \mathcal{T}_\perp$*;*
- $\mathbb{M}_{\mathsf{code}}$ *is a partial map that assigns to method names and class names a method body of the form* (x, c) *where x is a list of parameters such that x and* σ *have the same length, and c is either a Java statement, or a list of JVM bytecode.*

The set of programs is defined formally as

$$\begin{aligned} \mathit{Prog} &= \mathit{MethDecl} \times \mathit{ClassDecl} \\ \mathit{MethDecl} &= (\mathcal{M} \to \mathit{MethSig}) \times ((\mathcal{M} \times \mathcal{C}) \rightharpoonup \mathit{MethBody}) \\ \mathit{MethSig} &= \mathcal{T}^\star \times \mathcal{T}_\perp \\ \mathit{MethBody} &= \mathit{Var}^\star \times \mathit{Code} \\ \mathit{ClassDecl} &= (\mathcal{C} \times \mathcal{F}) \rightharpoonup \mathcal{T} \end{aligned}$$

Figures 1 and 2 define the fragment of Java and of the JVM considered. We use $\mathcal{X}$ to denote the set of variables and let x range over $\mathcal{X}$. We use n to range over integers, C to range over classes, and E to range over exception classes.

At the source level, we let $\mathcal{E}$ denote the set of expressions and s denote the set of statements. We assume that a code is a statement followed by a return expression, i.e. of the form $\mathcal{P} = s; \mathsf{return}\ e$ (note that returns may occur elsewhere in the statement). Note that expressions may have side-effects.

At the bytecode level, a code consists of a list of bytecode instructions, and a list of exception handlers, where an exception handler is a quadruple (b, e, C, i) where (b, e) is a pair of program points that define the range of the handler, i is the entry point of the handler, and E is the class of exceptions handled. We let j range over program points and assume that the code is well-formed, i.e. there is no jump outside the code.

instructions i ::=	push n	push value on top of stack
|	binop op	binary operation on stack
|	load x	load value of x on stack
|	store x	store top of stack in variable x
|	goto j	unconditional jump
|	if cmp j	conditional jump
|	return	return the top value of the stack
|	new C	instance creation
|	getfield f	field access
|	putfield f	field assignment
|	invoke m	virtual method call
|	throw	throw exception

Fig. 2. Java bytecode

Compilation. The compiler is defined in Figure 3 and constructs the exception handler function Handler simultaneously with the compilation of Java code to JVM code. To simplify the presentation the exception handler function is built incrementally (using imperative style). The only construct which affects the exception handler function is the try catch statement.

Provided the source and target languages are given appropriate semantics, it is possible to show that the compiler preserves the input/output semantics of programs. One interesting consequence of preservation of semantics, together with preservation of proof obligations, is that the soundness of the verification condition generators for Java and the JVM are inter-derivable, i.e. follow from each other.

4 Verification Condition Generation

We define verification condition generators for annotated Java and JVM programs. Both VC generators are direct style, i.e. they operate on the program itself, rather than on an intermediate representation.

Assertion language. The assertion language is shared between Java and JVM programs, and defined in Figure 4. As is common, only pure expressions can appear in assertions. Besides, the assertion language provides constructions to specify the heap, and in the

Compilation of expressions

$$
\begin{aligned}
k\colon\llbracket x\rrbracket &= k:\mathsf{load}\ x\\
k\colon\llbracket c\rrbracket &= k:\mathsf{push}\ c\\
k\colon\llbracket e_1\ op\ e_2\rrbracket &= k:\llbracket e_1\rrbracket;\ k_1:\llbracket e_2\rrbracket;\ k_2:\mathsf{binop}\ op\\
\text{where } k_1 &= k+|\llbracket e_1\rrbracket|\\
k_2 &= k_1+|\llbracket e_2\rrbracket|\\
k\colon\llbracket e.\mathsf{f}\rrbracket &= k:\llbracket e\rrbracket;k_1:\mathsf{getfield}\ \mathsf{f}\\
\text{where } k_1 &= k+|\llbracket e\rrbracket|\\
k\colon\llbracket \mathsf{new}\ \mathsf{C}\rrbracket &= k:\mathsf{new}\ \mathsf{C}\\
k\colon\llbracket e_1.\mathsf{m}(e_2)\rrbracket &= k:\llbracket e_1\rrbracket;k_1:\llbracket e_2\rrbracket;\ k_2:\mathsf{invoke}\ m\\
\text{where } k_1 &= k+|\llbracket e_1\rrbracket|\\
k_2 &= k_1+|\llbracket e_2\rrbracket|
\end{aligned}
$$

Compilation of statements

$$
\begin{aligned}
k\colon\llbracket x:=e\rrbracket &= k:\llbracket e\rrbracket;\ k_1:\ \mathsf{store}\ x\\
\text{where } k_1 &= k+|\llbracket e\rrbracket|\\
k\colon\llbracket s_1;s_2\rrbracket &= k\colon\llbracket s_1\rrbracket;k_2\colon\llbracket s_2\rrbracket\\
\text{where } k_2 &= k+|\llbracket s_1\rrbracket|\\
k\colon\llbracket \mathsf{return}\ e\rrbracket &= k:\ \llbracket e\rrbracket;k_1:\ \mathsf{return}\\
\text{where } k_1 &= k+|\llbracket e\rrbracket|\\
k\colon\llbracket \mathsf{if}(e_1\ cmp\ e_2)\{s_1\}\{s_2\}\rrbracket &= k:\llbracket e_1\rrbracket;k_1\ \llbracket e_2\rrbracket;k_2:\ \mathsf{if}\ cmp\ k_3+1;\\
&\quad\ k_2+1:\llbracket s_2\rrbracket;\ k_3:\mathsf{goto}\ l;\ k_3+1:\llbracket s_1\rrbracket\\
\text{where } k_1 &= k+|\llbracket e_1\rrbracket|\\
k_2 &= k_1+|\llbracket e_2\rrbracket|\\
k_3 &= k_2+|\llbracket s_2\rrbracket|+1\\
l &= k_3+|\llbracket s_1\rrbracket|+1\\
k\colon\llbracket \mathsf{while}(e_1\ cmp\ e_2)\{s\}\rrbracket &= k:\mathsf{goto}\ k_1;k+1:\llbracket s\rrbracket;\\
&\quad\ k_1:\llbracket e_1\rrbracket;\ k_2:\llbracket e_2\rrbracket;\ k_3:\mathsf{if}\ cmp\ k+1;\\
\text{where } k_1 &= k+1+|\llbracket s\rrbracket|\\
k_2 &= k_1+|\llbracket e_1\rrbracket|\\
k_3 &= k_2+|\llbracket e_2\rrbracket|\\
k\colon\llbracket e_1.\mathsf{f}:=e_2\rrbracket &= k:\llbracket e_1\rrbracket;\ k_1:\llbracket e_2\rrbracket;\ k_2:\mathsf{putfield}\ \mathsf{f}\\
\text{where } k_1 &= k+|\llbracket e_1\rrbracket|\\
k_2 &= k_1+|\llbracket e_2\rrbracket|\\
k\colon\llbracket \mathsf{throw}\ e\rrbracket &= k:\llbracket e\rrbracket;\ k_1:\mathsf{throw}\\
\text{where } k_1 &= k+|\llbracket e\rrbracket|\\
k\colon\llbracket \mathsf{try}\{s_1\}\ \mathsf{catch}\ (E)\ \{s_2\}\rrbracket &= k\colon\llbracket s_1\rrbracket;\mathsf{goto}\ \mathsf{k}_2;k_1\colon\llbracket s_2\rrbracket\\
\text{where } k_1 &= k+|\llbracket s_1\rrbracket|+1\\
k_2 &= k_1+|\llbracket s_2\rrbracket|\\
\mathsf{Handler}(j,E) &:= \begin{cases} k_1 & k\le j<k_1\ \wedge\ E\preceq E\\ \mathsf{Handler}(j,E) & \text{otherwise}\end{cases}
\end{aligned}
$$

Fig. 3. Definition of compiler

case of the JVM, the operand stack. More precisely, the assertion language features stack expressions, and heap expressions, as well as more traditional logical expressions.

We briefly comment on heap expressions: $\mathbf{h}$ is used to refer to the current heap, whereas $\bar{\mathbf{h}}$ refers to the initial heap—indeed it is useful that some assertions, such as invariants and post-conditions may refer to the initial state of the method. We use $H(\breve{e}.\mathsf{f})$ to represent the value of the field f of the expression $\breve{e}$ in the heap H. Finally, we let h be a heap variable, that will be introduced and universally bound in the weakest precondition for dynamic object creation.

Next, we briefly comment on stack expressions: $\mathbf{os}$ denotes the current operand stack, $\breve{os}[k]$ is used to refer to the k-th element of the operand stack, and $\uparrow^k \breve{os}$ denotes the stack obtained from $\breve{os}$ by popping its top k-elements. Note that stack expressions are not allowed in specifications, neither at the Java nor at the JVM level. However, the verification condition generator for the JVM will generate intermediate assertions that use these constructs.

Finally, observe that logical expressions also contain the special variable res, that is used to denote the result of executing the method, and the special variable exc, that is used to specify the termination mode of the method, as well as a construction $\bar{x}$ for every variable x, that is used to denote the original value stored in x at the onset of the method execution.

The definition of propositions is standard, except for the constructs $\mathsf{New}(H,C) = (h,v)$, and $\mathsf{typeof}(\breve{e}) \preceq C$. These constructs are used respectively to mimick at a logical level the creation of a new object in the heap, and in the exceptional postcondition of a *try-catch*.

$$
\begin{array}{lrcl}
\text{stack expressions} & \breve{os} & ::= & \mathbf{os} \mid \breve{e} :: \breve{os} \mid \uparrow^k \breve{os} \\
\text{logical expressions} & \breve{e} & ::= & \mathsf{res} \mid \bar{x} \mid x \mid v \mid c \mid \breve{e}\ op\ \breve{e} \mid H(\breve{e}.f) \mid \breve{os}[k] \\
\text{heap expressions} & H & ::= & H\{\breve{e}.f \mapsto \breve{e}\} \mid \bar{\mathbf{h}} \mid \mathbf{h} \mid h \\
\text{logical tests} & \breve{t} & ::= & \breve{e}\ cmp\ \breve{e} \\
\text{propositions} & P & ::= & \breve{t} \mid \neg P \mid P \wedge P \mid P \vee P \mid P \Rightarrow P \\
 & & & \mid \forall v.\ P \mid \forall h.\ P \mid \mathsf{New}(H,C) = (h,v) \mid \mathsf{typeof}(\breve{e}) \preceq C
\end{array}
$$

Fig. 4. Assertion language

Definition 2. *The set Assn of assertions is given in Figure 4. Assertions that do not refer to the operand stack are referred to as user assertions.*

– *The set $Assn_{\mathrm{local}}$ of local assertions is defined as the subset of user assertions that do not use* res*, and* exc.
– *The set $Assn_{\mathrm{pre}}$ of preconditions is defined as the subset of user assertions that do not use* res*, and* exc*, nor barred variables $\bar{x}$ and $\bar{\mathbf{h}}$.*
– *The set $Assn_{\mathrm{npost}}$ of normal postconditions is defined as the subset of user assertions that do not use* exc.
– *The set $Assn_{\mathrm{epost}}$ of exceptional postconditions are defined respectively as the set of user assertions that do not use* res.

Annotated programs Program annotations either specify the interface behavior of methods, or some internal behavior. More precisely, the interface behavior of each method is specified by a precondition and by (normal and exceptional) postconditions; these annotations, which are used for modular verification, are made accessible by other methods in a global specification table. In contrast, internal annotations are either required for effective generation of proof obligations, as is the case of loop invariants, or for enforcing non-functional properties of code, as is the case e.g. for ghost assignments to track usage policies.

Definition 3. *An annotated program is a triple (p, Γ, Λ), where p is a program, and Γ is a method specification table, and Λ is a local specification table for p, i.e.:*

- *Γ is a mapping from methods to method specifications, where a method specification consists of a precondition, of a normal postcondition, and of an exceptional postcondition.*
- *in the case of Java programs, Λ associates an invariant (a local assertion) to each while statement. We write $\mathsf{while}_I(t)\{e\}$ for stressing that I is an invariant for $\mathsf{while}(t)\{e\}$;*
- *in the case of JVM programs, Λ is a partial mapping from methods and program points to local assertions that should be understood as the precondition of the program point.*

The sets GST of method specification tables, LST of local specification table, and AnnotProg of annotated JVM programs are defined formally as:

$$\begin{aligned} AnnotProg &= Prog \times GST \times LST \\ GST &= \mathcal{M} \to Assn_{\text{pre}} \times Assn_{\text{npost}} \times Assn_{\text{epost}} \\ LMS &= \mathcal{M} \times PC \rightharpoonup Assn_{\text{local}} \end{aligned}$$

(The formal definition of annotated Java programs is similar and omitted.)

Verification condition generation for Java programs. The verification condition generator operates on annotated programs and returns a set of proof obligations. It relies on weakest precondition calculi for expressions and statements. The calculi are defined in Figure 5 and take as arguments two postconditions: a normal postcondition ψ which should be valid if the evaluation terminates normally and an exceptional postcondition which should be valid if the evaluation terminates exceptionally.

The first algorithm $\mathsf{wp}^{\mathsf{m}}_{\mathcal{E}}(e, \psi, \psi_e)_v$ computes the weakest precondition of the expression e. In a simple setting (without side-effects), $\mathsf{wp}^{\mathsf{m}}_{\mathcal{E}}(e, \psi, \psi_e)_v$ is simply $\psi\{v \mapsto e\}$, as shown by the first three rules for variables, constants and binary operators. The other rules reflect the imperative semantics of the expressions: e.g. for the new C expression two fresh variables h and v_1 are created, corresponding to the result of creating a new object v_1 of class C in the current heap $\mathbf{h}$ ($\mathsf{New}(\mathbf{h}, C) = (h, v_1)$), and the current heap of ψ is upgraded with the new heap h and the variable v by the new object v_1. The last two rules are more complicated due to the possibility of raising exceptions.

For field access e_1.f, if the evaluation of e_1 terminates normally, two cases are possible: either the value is not null and the heap is updated (condition ξ_1) or the value is null and a runtime exception (of class `NullPointerException`) is thrown. In the latter

Weakest precondition of expressions and comparisons

$$\frac{}{\mathsf{wp}^{\mathsf{m}}_{\mathcal{E}}(x,\psi,\psi_e)_v = \psi\{v \mapsto x\}} \qquad \frac{}{\mathsf{wp}^{\mathsf{m}}_{\mathcal{E}}(c,\psi,\psi_e)_v = \psi\{v \mapsto c\}}$$

$$\frac{}{\mathsf{wp}^{\mathsf{m}}_{\mathcal{E}}(e_1\ op\ e_2,\psi,\psi_e)_v = \mathsf{wp}^{\mathsf{m}}_{\mathcal{E}}(e_1,\mathsf{wp}^{\mathsf{m}}_{\mathcal{E}}(e_2,\psi\{v \mapsto v_1\ op\ v_2\},\psi_e)_{v_2},\psi_e)_{v_1}}$$

$$\frac{}{\mathsf{wp}^{\mathsf{m}}_{\mathcal{E}}(e_1\ cmp\ e_2,\psi,\psi_e)_v = \mathsf{wp}^{\mathsf{m}}_{\mathcal{E}}(e_1,\mathsf{wp}^{\mathsf{m}}_{\mathcal{E}}(e_2,\psi\{v \mapsto v_1\ cmp\ v_2\},\psi_e)_{v_2},\psi_e)_{v_1}}$$

$$\frac{}{\mathsf{wp}^{\mathsf{m}}_{\mathcal{E}}(\mathsf{new}\ C,\psi,\psi_e)_v = \forall\, h\, v_1.\ \mathsf{New}(\mathbf{h},C) = (h,v_1) \Rightarrow \psi\{\mathbf{h}, v \mapsto h, v_1\}}$$

$$\frac{\begin{array}{c}\xi_1 := v_1 \neq \mathsf{null} \Rightarrow \psi\{v \mapsto \mathbf{h}(v_1.\mathsf{f})\}\\ \xi_2 := \forall h\ r,\ v_1 = \mathsf{null} \Rightarrow \mathsf{NPE}(\mathbf{h},h,r) \Rightarrow \psi_e(h,r)\end{array}}{\mathsf{wp}^{\mathsf{m}}_{\mathcal{E}}(e_1.\mathsf{f},\psi,\psi_e)_v = \mathsf{wp}^{\mathsf{m}}_{\mathcal{E}}(e_1,\xi_1 \wedge \xi_2,\psi_e)_{v_1}}$$

$$\frac{\begin{array}{c}\phi_n := \forall h\ r,\ \Psi(v_1,v_2,\mathbf{h},h,r) \Rightarrow \psi\{\mathbf{h}, v \mapsto h, r\} \qquad \Gamma(\mathsf{m}',\Phi,\Psi,\Psi_e)\\ \phi_e := \forall h\ r,\ \Psi_e(v_1,v_2,\mathbf{h},h,r) \Rightarrow \psi_e(h,r)\\ \xi_1 := v_1 \neq \mathsf{null} \Rightarrow \Phi(v_1,v_2,\mathbf{h}) \wedge \phi_n \wedge \phi_e\\ \xi_2 := v_1 = \mathsf{null} \Rightarrow \forall h\ r,\ \mathsf{NPE}(\mathbf{h},h,r) \Rightarrow \psi_e(h,r)\end{array}}{\mathsf{wp}^{\mathsf{m}}_{\mathcal{E}}(e_1.\mathsf{m}'(e_2),\psi,\psi_e)_v = \mathsf{wp}^{\mathsf{m}}_{\mathcal{E}}(e_1,\mathsf{wp}^{\mathsf{m}}_{\mathcal{E}}(e_2,\xi_1 \wedge \xi_2,\psi_e)_{v_2},\psi_e)_{v_1}}$$

Weakest precondition of statements

$$\frac{\Gamma(\mathsf{m},\Phi,\Psi,\Psi_e)}{\mathsf{wp}^{\mathsf{m}}_{\mathcal{E}}(\mathsf{return}\ e,\psi,\psi_e) = \mathsf{wp}^{\mathsf{m}}_{\mathcal{E}}(e,\Psi\{\mathsf{res} \mapsto v\},\psi_e)}$$

$$\frac{}{\mathsf{wp}^{\mathsf{m}}_{\mathcal{E}}(x := e,\psi,\psi_e) = \mathsf{wp}^{\mathsf{m}}_{\mathcal{E}}(e,\psi,\psi_e)_x, \emptyset}$$

$$\frac{(\phi_2,\theta_2) := \mathsf{wp}^{\mathsf{m}}_{\mathcal{E}}(s_2,\psi,\psi_e) \qquad (\phi_1,\theta_1) := \mathsf{wp}^{\mathsf{m}}_{\mathcal{E}}(s_1,\phi_2,\psi_e)}{\mathsf{wp}^{\mathsf{m}}_{\mathcal{E}}(s_1;\ s_2,\psi,\psi_e) = \phi_1, \theta_1 \cup \theta_2}$$

$$\frac{(\phi_1,\theta_1) := \mathsf{wp}^{\mathsf{m}}_{\mathcal{E}}(s_1,\psi,\psi_e) \qquad (\phi_2,\theta_2) := \mathsf{wp}^{\mathsf{m}}_{\mathcal{E}}(s_2,\psi,\psi_e)}{\mathsf{wp}^{\mathsf{m}}_{\mathcal{E}}(\mathsf{if}(t)\{s_1\}\{s_2\},\psi,\psi_e) = \mathsf{wp}^{\mathsf{m}}_{\mathcal{E}}(t, v \Rightarrow \phi_1 \wedge \neg v \Rightarrow \phi_2,\psi_e)_v, \theta_1 \cup \theta_2}$$

$$\frac{(\xi_1,\theta) := \mathsf{wp}^{\mathsf{m}}_{\mathcal{E}}(s,I,\psi_e) \qquad \xi_2 := \mathsf{wp}^{\mathsf{m}}_{\mathcal{E}}(t, v \Rightarrow \xi_1 \wedge \neg v \Rightarrow \psi,\psi_e)_v}{\mathsf{wp}^{\mathsf{m}}_{\mathcal{E}}(\mathsf{while}_I(t)\{s\},\psi,\psi_e) = I, I \Rightarrow \xi_2 \cup \theta}$$

$$\frac{\begin{array}{c}\xi_1 := v_1 \neq \mathsf{null} \Rightarrow \psi\{\mathbf{h} \mapsto \mathbf{h}\{v_1.\mathsf{f} \mapsto v_2\}\}\\ \xi_2 := \forall h\ r,\ v_1 = \mathsf{null} \Rightarrow \mathsf{NPE}(\mathbf{h},h,r) \Rightarrow \psi_e(h,r)\end{array}}{\mathsf{wp}^{\mathsf{m}}_{\mathcal{E}}(e_1.\mathsf{f} := e_2,\psi,\psi_e) = \mathsf{wp}^{\mathsf{m}}_{\mathcal{E}}(e_1,\mathsf{wp}^{\mathsf{m}}_{\mathcal{E}}(e_2,\xi_1 \wedge \xi_2,\psi_e)_{v_2},\psi_e)_{v_1}, \emptyset}$$

$$\frac{\begin{array}{c}\xi_1 := v \neq \mathsf{null} \Rightarrow \psi_e(\mathbf{h},v)\\ \xi_2 := \forall h\ r,\ v = \mathsf{null} \Rightarrow \mathsf{NPE}(\mathbf{h},h,r) \Rightarrow \psi_e(h,r)\end{array}}{\mathsf{wp}^{\mathsf{m}}_{\mathcal{E}}(\mathsf{throw}\ e,\psi,\psi_e) = \mathsf{wp}^{\mathsf{m}}_{\mathcal{E}}(e,\xi_1 \wedge \xi_2,\psi_e)_v, \emptyset}$$

$$\frac{(\psi',\theta_2) := \mathsf{wp}^{\mathsf{m}}_{\mathcal{E}}(s_2,\psi,\psi_e) \qquad (\phi,\theta_1) := \mathsf{wp}^{\mathsf{m}}_{\mathcal{E}}(s_1,\psi,(E,\psi') \otimes \psi_e)}{\mathsf{wp}^{\mathsf{m}}_{\mathcal{E}}(\mathsf{try}\{s_1\}\ \mathsf{catch}\ (E)\ \{s_2\},\psi,\psi_e) = \phi, \theta_1 \cup \theta_2}$$

Fig. 5. VCGEN for Java

case the postcondition is the exceptional postcondition (condition ξ_2 where $\mathsf{NPE}(\mathbf{h}, h, r)$ is a shortcut for $\mathsf{New}(\mathbf{h}, \texttt{NullPointerException}) = (h, r)$ and $\psi_e(h, r)$ a shortcut for $\psi_e\{\mathbf{h}, \mathsf{exc} \mapsto h, r\}$).

For method invocation $e_1.\mathsf{m}'(e_2)$, if the evaluation of the two expressions e_1 and e_2 terminate normally, two possibilities arise: either e_1 evaluates to null and a null pointer exception is thrown (condition ξ_2) or e_1 evaluates to a non-null reference and the call is performed, and we must ensure that the precondition is valid. Again there are two possibilities: either the method m' throws an exception r, in which case we should guarantee that the exceptional postcondition is valid, under the assumption that the exceptional postcondition of the method m' holds

$$\Psi_e(v_1, v_2, \mathbf{h}, h, r) = \Psi_e\{\bar{\mathsf{this}}, \bar{arg}, \bar{\mathbf{h}}, \mathbf{h}, \mathsf{exc} \mapsto v_1, v_2, \mathbf{h}, h, r\}$$

or the method m' terminates normally, in which case we should guarantee the normal postcondition is valid, under the assumption that the normal postcondition of m' holds

$$\Psi(v_1, v_2, \mathbf{h}, h, r) = \Psi\{\bar{\mathsf{this}}, \bar{arg}, \bar{\mathbf{h}}, \mathbf{h}, \mathsf{res} \mapsto v_1, v_2, \mathbf{h}, h, r\}$$

The second algorithm $\mathsf{wp}^{\mathsf{m}}_{\mathcal{L}}(s, \psi, \psi_e)$ returns the weakest precondition of s and a set of proof obligations. We focus on the cases of the *while* and *try-catch* statements, that are the most interesting.

The *while* statement is the only statement generating a proof obligation. Its precondition is simply its loop invariant I which should be valid before the loop. The proof obligation $I \Rightarrow \xi_2$ conjointly with the proof obligations of θ ensures that under the validity of the invariant I. If the test evaluates normally to `true` then the precondition of the loop body is satisfied and if the test evaluates normally to `false` then the postcondition of the loop is satisfied. Furthermore, if the evaluation of the test terminates abruptly then the exceptional postcondition is satisfied. Note that the precondition of *while* statement is defined in a slightly different way than usual, in the style of *if* statement, generating only one proof obligation, in order to ensure preservation of proof obligations. This is due to the fact that at bytecode level there are no loops but only conditional jumps.

The *try-catch* statement is the only instruction modifying the exceptional postcondition. The weakest precondition of s_1 is not evaluated with exceptional postcondition ψ_e, since if s_1 throws an exception which is a subclass of E the exception is caught by the *catch* and s_2 is executed. To reflect this, the weakest precondition algorithm first computes the precondition ψ' of s_2 and uses it in the exceptional postcondition of s_1:

$$(E, \psi') \otimes \psi_e = (\mathsf{typeof}(\mathsf{exc}) \preceq E \Rightarrow \psi') \wedge (\mathsf{typeof}(\mathsf{exc}) \not\preceq E \Rightarrow \psi_e)$$

The precondition of the *try-catch* statement is the precondition of s_1.

The set of proof obligations of a method m is defined by

$$\frac{\Gamma(\mathsf{m}, \Phi, \Psi, \Psi_e) \quad p[m] = s; \mathsf{return}\ e \qquad \mathsf{wp}^{\mathsf{m}}_{\mathcal{L}}(s, \mathsf{wp}^{\mathsf{m}}_{\mathcal{L}}(e, \Psi\{\mathsf{res} \mapsto v\}, \Psi_e)_v, \Psi_e) = \phi, \theta}{\mathsf{PO}^{\mathsf{java}}(p, \Gamma, \mathsf{m}) = \{\Phi \Rightarrow \phi\{\bar{\mathbf{h}}, \bar{\mathsf{this}}, \bar{arg} \mapsto \mathbf{h}, \mathsf{this}, arg\}\} \cup \theta}$$

In other words, the proof obligations of the method body should be valid and the precondition of the method should imply the precondition of its body. Then, the proof obligations of a class C in a program p is defined by the union of the proof obligations of the method defined in C, and finally the proof obligations of a Java program p is the union of the proof obligation of each class of the program:

$$\begin{aligned}\mathsf{PO}^{\mathsf{java}}(p,\Gamma,C) &= \cup_{\mathsf{m}\in C}\mathsf{PO}^{\mathsf{java}}(p,\Gamma,\mathsf{m})\\ \mathsf{PO}^{\mathsf{java}}(p,\Gamma) &= \cup_{C\in p}\mathsf{PO}^{\mathsf{java}}(p,\Gamma,C)\end{aligned}$$

Verification condition generation for JVM programs The verification of JVM programs operates on JVM programs that carry sufficient annotations, and produces a fully annotated program in which all labels of the program have an explicit precondition attached to them. Formally, the property of being sufficiently annotated is characterized by an inductive and decidable definition and does not impose any specific structure on programs. The definition uses the successor relation between program points, which is defined in the usual way. Given k, we let $\mathsf{succ}_{\dot{p},\mathsf{m}}(k)$ denotes its set of successors.

Definition 4 (Sufficiently annotated program).

– *A program point k' is reachable from a label k in* m *and $\dot{p}$ if $k = k'$ or if k' is the successor of a program point reachable from k:*

$$\frac{}{k \in \mathsf{reachable}_{\dot{p},\mathsf{m},k}} \qquad \frac{k' \in \mathsf{reachable}_{\dot{p},\mathsf{m},k} \quad k'' \in \mathsf{succ}_{\dot{p},\mathsf{m}}(k')}{k'' \in \mathsf{reachable}_{\dot{p},\mathsf{m},k}}$$

– *Given a bytecode program $\dot{p}$ and a local annotation table Λ, a program point k in* m *reaches annotated program points if the instruction at position k is annotated in* Λ_{m} *or if the instruction is a* return *(in that case the annotation is the postcondition), or if all its immediate successors reach annotated program points. More precisely,* $\mathsf{reachA}_{\dot{p},\Lambda,\mathsf{m}}$ *is defined as the smallest set that satisfies the following conditions:*

$$\frac{\Lambda_{\mathsf{m}}(k) = \dot{P}}{k \in \mathsf{reachA}_{\dot{p},\Lambda,\mathsf{m}}} \qquad \frac{\dot{p}_{\mathsf{m}}[k] = \mathsf{return}}{k \in \mathsf{reachA}_{\dot{p},\Lambda,\mathsf{m}}} \qquad \frac{\forall k' \in \mathsf{succ}_{\dot{p},\mathsf{m}}(k), k' \in \mathsf{reachA}_{\dot{p},\Lambda,\mathsf{m}}}{k \in \mathsf{reachA}_{\dot{p},\Lambda,\mathsf{m}}}$$

– *A method* m *in a program $\dot{p}$ and a local annotation table Λ is sufficiently annotated if every reachable point from the starting point (i.e., label 0) reaches annotated labels.*
– *An annotated program $(\dot{p},\Gamma,\Lambda)$ is sufficiently annotated if all methods* m *in $\dot{p}$ is sufficiently annotated.*

Given a sufficiently annotated program $(\dot{p},\Gamma,\Lambda)$, the verification condition generator is defined with three mutually recursive functions $\mathsf{wp}_i^{\mathsf{m}}(k)$, $\mathsf{wp}_l^{\mathsf{m}}(k)$ and $\mathsf{wp}_e^{\mathsf{m}}(k)$ defined in Figure 6. The function $\mathsf{wp}_l^{\mathsf{m}}(k)$ computes the weakest precondition of the program point k in a method m using the local annotation table Λ. If k is annotated ($\Lambda_{\mathsf{m}}(k) = \dot{P}$), the weakest precondition is the annotation $\dot{P}$; otherwise the weakest precondition is $\mathsf{wp}_i^{\mathsf{m}}(k)$. The function $\mathsf{wp}_i^{\mathsf{m}}(k)$ is really the predicate transformer: it first computes the weakest precondition of all the successors of the instruction at label k and then transforms the resulting conditions depending on the instruction. The last function ($\mathsf{wp}_e^{\mathsf{m}}(k)$) computes the weakest precondition of an instruction in the case where its evaluation leads to an exception.

We explain how the computations are performed for selected instructions.

If the instruction at position k is $\mathsf{load}\ x$, the instruction pushes the value of x on the top of the operand stack and its precondition $\mathsf{wp}_i^{\mathsf{m}}(k)$ is the precondition of the instruction at $k+1$ where the current operand stack $\mathbf{os}$ is replaced by operand stack after the evaluation of $\mathsf{load}\ x$ ($x :: \mathbf{os}$).

If the instruction is $\mathsf{store}\ x$ the top value of the stack is stored in x and removed from the stack, so its precondition is the precondition of its successor $\mathsf{wp}_l^{\mathsf{m}}(k+1)$ where $\mathbf{os}$ is popped by one ($\uparrow \mathbf{os}$) and x is replaced by the top of the stack ($\mathbf{os}[0]$).

A more interesting instruction is the $\mathsf{getfield}\ \mathsf{f}$ which can raise a runtime exception. As for Java programs, the precondition is the conjunction of the two possible executions: if the first element of the stack $\mathbf{os}[0]$ is not null then the execution access to the field value and pushes it on the top of the stack where the first element has been removed; otherwise the first element is null and an runtime exception is caught, so the weakest precondition uses the function $\mathsf{wp}_e^{\mathsf{m}}$.

Note that the definition of $\mathsf{wp}_e^{\mathsf{m}}$ strongly depends on the exception handler of m and thus preliminary explanations are in order. Given a list L of pairs (exception class, program point), let $H(E,L)$ be the following partial function:

$$\begin{array}{ll} H(E,\emptyset) & = \bot \\ H(E,(E_1,j_1) :: L) & = j_1 \text{ if } E \preceq E_1 \\ H(E,(E_1,j_1) :: L) & = H(E,L) \text{ if } E \not\preceq E_1 \end{array}$$

Intuitively, the $H(E,L)$ compute the next program point j (if exists) if a exception of class E is thrown. Given a program point k in m the exception handler can be see as a list of pairs $L_k = (E_1,j_1) :: \ldots :: (E_n,j_n) :: \emptyset$ satisfying $\mathsf{Handler}(k,E) = H(L_k)$. The function $\mathsf{wp}_e^{\mathsf{m}}(k)$ is defined as $H_{\mathsf{wp}}^{\mathsf{m}}(L_k)$ where $H_{\mathsf{wp}}^{\mathsf{m}}$ is the following function

$$\begin{array}{ll} H_{\mathsf{wp}}^{\mathsf{m}}(\emptyset) & = \Psi_e \text{ if } \Gamma(\mathsf{m},\Phi,\Psi,\Psi_e) \\ H_{\mathsf{wp}}^{\mathsf{m}}((E_1,j_1) :: L) & = (E_1,\mathsf{wp}_l^{\mathsf{m}}(j_1)) \otimes H_{\mathsf{wp}}^{\mathsf{m}}(L) \end{array}$$

Furthermore when an exception is thrown the operand stack is cleared. So the notation $\mathsf{wp}_e^{\mathsf{m}}(k,h,r)$ is a shortcut for $\mathsf{wp}_e^{\mathsf{m}}(k)\{\mathbf{h},\mathbf{os},\mathsf{exc} \mapsto h,\emptyset,r\}$.

The proof obligations of a method m is defined by

$$\frac{\Gamma(\mathsf{m},\Phi,\Psi,\Psi_e) \quad \theta = \cup_{\{k \mid \Lambda_{\mathsf{m}}(k)=\dot{P}\}} \mathsf{wp}_l^{\mathsf{m}}(k) \Rightarrow \mathsf{wp}_i^{\mathsf{m}}(k)\}}{\mathsf{PO}^{\mathsf{bc}}(\dot{p},\Gamma,\Lambda,\mathsf{m}) = \{\Phi \Rightarrow \mathsf{wp}_i^{\mathsf{m}}(0)\{\bar{\mathbf{h}},\bar{\mathsf{this}},\bar{arg} \mapsto \mathbf{h},\mathsf{this},arg\}\} \cup \theta}$$

Then the proof obligations for a class and a program are defined in the same way that for Java.

Soundness issues. The verification condition generators are sound, in the sense that programs whose verification conditions are provable meet their specifications, provided we impose some additional restrictions on annotated programs. For example, we must assume some form of coherence between the specification of methods at different classes in a hierarchy. This can be achieved e.g. using ideas of behavioral subtyping. In addition, we informally rely on some properties of typing, by proving soundness w.r.t. a defensive semantics.

Weakest precondition of instructions

$$\frac{\dot{p}_{\mathsf{m}}[k] = \mathsf{binop}\ op}{\mathsf{wp}_i^{\mathsf{m}}(k) = \mathsf{wp}_l^{\mathsf{m}}(k+1)\{\mathbf{os} \mapsto (\mathbf{os}[1]\ op\ \mathbf{os}[0]) :: \uparrow^2 \mathbf{os}\}}$$

$$\frac{\dot{p}_{\mathsf{m}}[k] = \mathsf{load}\ x}{\mathsf{wp}_i^{\mathsf{m}}(k) = \mathsf{wp}_l^{\mathsf{m}}(k+1)\{\mathbf{os} \mapsto x :: \mathbf{os}\}} \qquad \frac{\dot{p}_{\mathsf{m}}[k] = \mathsf{store}\ x}{\mathsf{wp}_i^{\mathsf{m}}(k) = \mathsf{wp}_l^{\mathsf{m}}(k+1)\{\mathbf{os}, x \mapsto \uparrow \mathbf{os}, \mathbf{os}[0]\}}$$

$$\frac{\dot{p}_{\mathsf{m}}[k] = \mathsf{push}\ c}{\mathsf{wp}_i^{\mathsf{m}}(k) = \mathsf{wp}_l^{\mathsf{m}}(k+1)\{\mathbf{os} \mapsto c :: \mathbf{os}\}} \qquad \frac{\dot{p}_{\mathsf{m}}[k] = \mathsf{goto}\ l}{\mathsf{wp}_i^{\mathsf{m}}(k) = \mathsf{wp}_l^{\mathsf{m}}(l)}$$

$$\frac{\dot{p}_{\mathsf{m}}[k] = \mathsf{return} \quad \Gamma(m, \Phi, \Psi, \Psi_e)}{\mathsf{wp}_i^{\mathsf{m}}(k) = \Psi\{\mathsf{res} \mapsto \mathbf{os}[0]\}}$$

$$\frac{\dot{p}_{\mathsf{m}}[k] = \mathsf{if}\ cmp\ l \quad \xi_1 = \mathsf{wp}_l^{\mathsf{m}}(k+1)\{\mathbf{os} \mapsto \uparrow^2 \mathbf{os}\} \quad \xi_2 = \mathsf{wp}_l^{\mathsf{m}}(l)\{\mathbf{os} \mapsto \uparrow^2 \mathbf{os}\}}{\mathsf{wp}_i^{\mathsf{m}}(k) = (\mathbf{os}[1]\ cmp\ \mathbf{os}[0] \Rightarrow \xi_2) \wedge (\neg(\mathbf{os}[1]\ cmp\ \mathbf{os}[0]) \Rightarrow \xi_1)}$$

$$\frac{\begin{array}{c}\dot{p}_{\mathsf{m}}[k] = \mathsf{throw} \quad \xi_1 := \mathbf{os}[0] \neq \mathsf{null} \Rightarrow \mathsf{wp}_e^{\mathsf{m}}(k, \mathbf{h}, \mathbf{os}[0]) \\ \xi_2 := \forall h\ r,\ \mathbf{os}[0] = \mathsf{null} \Rightarrow \mathsf{NPE}(\mathbf{h}, h, r) \Rightarrow \mathsf{wp}_e^{\mathsf{m}}(k, h, r)\end{array}}{\mathsf{wp}_i^{\mathsf{m}}(k) = \xi_1 \wedge \xi_2}$$

$$\frac{\begin{array}{c}\dot{p}_{\mathsf{m}}[k] = \mathsf{putfield}\ \mathsf{f} \\ \xi_1 := \mathbf{os}[1] \neq \mathsf{null} \Rightarrow \mathsf{wp}_l^{\mathsf{m}}(k+1)\{\mathbf{h}, \mathbf{os} \mapsto \mathbf{h}\{\mathbf{os}[1].\mathsf{f} \mapsto \mathbf{os}[0]\}, \uparrow^2 \mathbf{os}\} \\ \xi_2 := \forall h\ r,\ \mathbf{os}[1] = \mathsf{null} \Rightarrow \mathsf{NPE}(\mathbf{h}, h, r) \Rightarrow \mathsf{wp}_e^{\mathsf{m}}(k, h, r)\end{array}}{\mathsf{wp}_i^{\mathsf{m}}(k) = \xi_1 \wedge \xi_2}$$

$$\frac{\begin{array}{c}\dot{p}_{\mathsf{m}}[k] = \mathsf{getfield}\ \mathsf{f} \quad \xi_1 := \mathbf{os}[0] \neq \mathsf{null} \Rightarrow \mathsf{wp}_l^{\mathsf{m}}(k+1)\phi\{\mathbf{os} \mapsto \mathbf{h}(\mathbf{os}[0].\mathsf{f}) :: \uparrow \mathbf{os}\} \\ \xi_2 := \forall h\ r,\ \mathbf{os}[0] = \mathsf{null} \Rightarrow \mathsf{NPE}(\mathbf{h}, h, r) \Rightarrow \mathsf{wp}_e^{\mathsf{m}}(k, h, r)\end{array}}{\mathsf{wp}_i^{\mathsf{m}}(k) = \xi_1 \wedge \xi_2}$$

$$\frac{\dot{p}_{\mathsf{m}}[k] = \mathsf{new}\ \mathsf{C}}{\mathsf{wp}_i^{\mathsf{m}}(k) = \forall h\ r.\ \mathsf{New}(\mathbf{h}, \mathsf{C}) = (h, r) \Rightarrow \mathsf{wp}_l^{\mathsf{m}}(k+1)\{\mathbf{h}, \mathbf{os} \mapsto h, r :: \mathbf{os}\}}$$

$$\frac{\begin{array}{c}\dot{p}_{\mathsf{m}}[k] = \mathsf{invoke}\ \mathsf{m}' \qquad \Gamma(\mathsf{m}', \Phi, \Psi, \Psi_e) \\ \phi_n := \forall h\ r,\ \Psi(\mathbf{os}[1], \mathbf{os}[0], r, \mathbf{h}, h) \Rightarrow \mathsf{wp}_l^{\mathsf{m}}(k+1)\{\mathbf{os}, \mathbf{h} \mapsto r :: \uparrow \mathbf{os}, h\} \\ \phi_e := \forall h\ r,\ \Psi_e(\mathbf{os}[1], \mathbf{os}[0], \mathbf{h}, h, r) \Rightarrow \mathsf{wp}_e^{\mathsf{m}}(k, h, r) \\ \xi_1 := \mathbf{os}[1] \neq \mathsf{null} \Rightarrow \Phi(\mathbf{os}[1], \mathbf{os}[0], \mathbf{h}) \wedge \phi_n \wedge \phi_e \\ \xi_2 := \mathbf{os}[1] = \mathsf{null} \Rightarrow \forall h\ r,\ \mathsf{NPE}(\mathbf{h}, h, r) \Rightarrow \mathsf{wp}_e^{\mathsf{m}}(k, h, r)\end{array}}{\mathsf{wp}_i^{\mathsf{m}}(i) = \xi_1 \wedge \xi_2}$$

Weakest precondition of instructions using the local annotation table

$$\frac{\Lambda_{\mathsf{m}}(k) = \dot{P}}{\mathsf{wp}_l^{\mathsf{m}}(k) = \dot{P}} \qquad \frac{\Lambda_{\mathsf{m}}(k) = \perp}{\mathsf{wp}_l^{\mathsf{m}}(k) = \mathsf{wp}_i^{\mathsf{m}}(k)}$$

Fig. 6. VCGEN for Java Bytecode

5 Preservation of Proof Obligations

The main result of this paper is that proof obligations are preserved by compilation; in other words, the proof obligations attached to an annotated program are syntactically equal to the proof obligations attached to its compilation. In order to state this result formally, we must first extend the compiler to annotated programs. Since method specification tables remain unchanged, it is only necessary to generate local method tables from annotated source statements.

During the compilation of a method m, only the compilation of *while* instructions has to be extended:

$$\begin{aligned} k : [\![\mathsf{while}_I(e_1\ cmp\ e_2)\{s\}]\!] &= k : \mathsf{goto}\ k_1; k+1 : [\![s]\!];\\ &\quad k_1 : [\![e_2]\!];\ k_2 : [\![e_1]\!];\ k_3 : \mathsf{if}\ cmp\ k+1;\\ \text{and } \Lambda_{\mathsf{m}}(k_1) &= I \end{aligned}$$

So the loop invariant is set just before starting the evaluation of the test. Note that, under such a definition, compiled programs are sufficiently annotated and thus the verification condition generator for JVM programs can be applied to them.

The proof of preservation of proof obligations proceeds in a standard way (for example, proof of preservation of semantics of a compiler is following the same pattern): we prove the result for expressions, then for statements, and finally for programs.

Lemma 1 (PPO for expressions). *For all expressions e and stack expressions* st *and program counters k and variables v such that $\dot{p}_{\mathsf{m}}[k..j-1] = k : [\![e]\!]$ and v does not appear in* st*, the following equality holds:*

$$\mathsf{wp}^{\mathsf{m}}_{\mathcal{E}}(e, \mathsf{wp}^{\mathsf{m}}_{l}(j)\{\mathbf{os} \mapsto v :: \mathsf{st}\}, \mathsf{wp}^{\mathsf{m}}_{e}(k))_v = \mathsf{wp}^{\mathsf{m}}_{i}(k)\{\mathbf{os} \mapsto \mathsf{st}\}$$

Proof *by induction on e. It is important to notice that the exception handler is the same for all $k \le i < j$. Let $\psi_e = \mathsf{wp}^{\mathsf{m}}_{e}(k)$. We only consider the case where $e = e_1\ op\ e_2$. Then, we have $k : [\![e]\!] = k : [\![e_1]\!]; k_1 : [\![e_2]\!]; k_2 : \mathsf{binop}\ op$ and $j = k_2 + 1$.*

$$\begin{aligned} &\mathsf{wp}^{\mathsf{m}}_{\mathcal{E}}(e_1\ op\ e_2, \mathsf{wp}^{\mathsf{m}}_{l}(k_2+1)\{\mathbf{os} \mapsto v :: \mathsf{st}\}, \psi_e)_v = \\ &\mathsf{wp}^{\mathsf{m}}_{\mathcal{E}}(e_1, \mathsf{wp}^{\mathsf{m}}_{\mathcal{E}}(e_2, \mathsf{wp}^{\mathsf{m}}_{l}(k_2+1)\{\mathbf{os} \mapsto v_1\ op\ v_2 :: \mathsf{st}\}, \psi_e)_{v_2}, \psi_e)_{v_1} \end{aligned}$$

Since $\mathsf{wp}^{\mathsf{m}}_{l}(k_2) = \mathsf{wp}^{\mathsf{m}}_{l}(k_2+1)\{\mathbf{os} \mapsto \mathbf{os}[1]\ op\ \mathbf{os}[0] :: \uparrow^2 \mathbf{os}\}$ we have

$$\mathsf{wp}^{\mathsf{m}}_{l}(k_2)\{\mathbf{os} \mapsto v_2 :: v_1 :: \mathsf{st}\} = \mathsf{wp}^{\mathsf{m}}_{l}(k_2+1)\{\mathbf{os} \mapsto v_1\ op\ v_2 :: \mathsf{st}\}$$

By induction hypothesis on e_2 we get

$$\mathsf{wp}^{\mathsf{m}}_{\mathcal{E}}(e_2, \mathsf{wp}^{\mathsf{m}}_{l}(k_2)\{\mathbf{os} \mapsto v_2 :: v_1 :: \mathsf{st}\}, \psi_e)_{v_2} = \mathsf{wp}^{\mathsf{m}}_{i}(k_1)\{\mathbf{os} \mapsto v_1 :: stack\}$$

Since $\Lambda_{\mathsf{m}}(k_1) = \bot$ we have $\mathsf{wp}^{\mathsf{m}}_{l}(k_1) = \mathsf{wp}^{\mathsf{m}}_{i}(k_1)$, and we conclude using the induction hypothesis on e_1.

Lemma 2 (PPO for statements). *For all statement s in the method body of* m *such that $\dot{p}_{\mathsf{m}}[k..j-1] = k : [\![s]\!]$. Let $(\phi, \theta) = \mathsf{wp}^{\mathsf{m}}_{\mathcal{E}}(s, \mathsf{wp}^{\mathsf{m}}_{l}(j), \mathsf{wp}^{\mathsf{m}}_{e}(k))$ then $\mathsf{wp}^{\mathsf{m}}_{l}(k) = \phi$ and for all P, $P \in \theta$ iff there exists l and I such that $k \le l < j$ and $\Lambda_{\mathsf{m}}(i) = I$ and $P = I \Rightarrow \mathsf{wp}^{\mathsf{m}}_{i}(l)$*

Proof *by induction on s. We only consider the case when* $s = \mathsf{while}_I(e_1\ cmp\ e_2)\{s_1\}$. *Let* $\psi = \mathsf{wp}_l^{\mathsf{m}}(k_3+1)$ *and* $\psi_e = \mathsf{wp}_e^{\mathsf{m}}(k)$ *and* $t = e_1\ cmp\ e_2$. *Then we have*

$$\begin{aligned} k : [\![\mathsf{while}_I(e_1\ cmp\ e_2)\{s\}]\!] = k &: \mathsf{goto}\ k_1; k+1 : [\![s]\!]; \\ & k_1 : [\![e_1]\!];\ k_2 : [\![e_2]\!];\ k_3 : \mathsf{if}\ cmp\ k+1 \end{aligned}$$

and $\Lambda_{\mathsf{m}}(k_1) = I$, *and* $(\xi_1, \theta_1) = \mathsf{wp}_{\mathcal{I}}^{\mathsf{m}}(s_1, I, \psi_e)$, *and* $\xi_2 = \mathsf{wp}_{\mathcal{I}}^{\mathsf{m}}(t, v \Rightarrow \xi_1 \wedge \neg v \Rightarrow \psi, \psi_e)_v$, *and* $\phi = I$, *and* $\theta = \{I \Rightarrow \xi_2\} \cup \theta_1$. *Since only* k_1 *is annotated* $\mathsf{wp}_l^{\mathsf{m}}(k) = \mathsf{wp}_i^{\mathsf{m}}(k) = \mathsf{wp}_l^{\mathsf{m}}(k_1) = I = \phi$.

We must prove the iff part of the lemma. By induction hypothesis we know that the property holds for all P is in θ_1, *so we should prove it for* $P = I \Rightarrow \xi_2$. *Using the induction hypothesis we have* $\xi_1 = \mathsf{wp}_l^{\mathsf{m}} k+1$, *by definition*

$$\begin{aligned} \mathsf{wp}_l^{\mathsf{m}}(k_3) = {} & (\mathbf{os}[1]\ cmp\ \mathbf{os}[0] \Rightarrow \mathsf{wp}_l^{\mathsf{m}}(k+1)\{\mathbf{os} \mapsto \uparrow^2 \mathbf{os}\}) \wedge \\ & (\mathbf{os}[1]\ cmp\ \mathbf{os}[0] \Rightarrow \psi\{\mathbf{os} \mapsto \uparrow^2 \mathbf{os}\}) \end{aligned}$$

Using Lemma 1 with $\mathsf{st} = v_2 :: v_1 :: \emptyset$ *we prove* $\mathsf{wp}_l^{\mathsf{m}}(k_1) = \xi_2$ *which conclude the proof.*

This lemma ensures that the side conditions generated for a method at source and bytecode level are the same.

Theorem 1. *Let p be an annotated program, and* Γ *be a specification table for p.*

$$\mathsf{PO}^{\mathrm{java}}(p, \Gamma) = \mathsf{PO}^{\mathrm{bc}}([\![p]\!], \Gamma)$$

6 Practical Issues

Theorem 1 proves preservation of proof obligations for an idealization of the Java Virtual Machine and of the Java compiler. In reality, preservation of proof obligations "almost holds". The purpose of this section is to briefly comment on the main issues that arise in a more realistic setting.

First, our verification condition generators are based on an idealized operational semantics of Java and the JVM; e.g. both semantics omit features such as constructors. It is possible to modify the verification condition generators to account for these features, while maintaining preservation of proof obligations. On the other hand, the Java Virtual Machine does not manipulate variable names, but indexes. Therefore, preservation of proof obligations only holds in practice up to renaming. In an implementation, the renaming of variables into certificates for source programs can be achieved without much effort by using compiler information to track the relation between variables and indexes; by this means, one can obtain certificates for bytecode programs.

Second, the JVM supports a subset of the Java basic types: for example, booleans only exist at the level of the Java language, and are compiled into integers. To preserve provability of specifications, we resort to standard techniques of interpreting typed predicate logic into predicate logic. Essentially, it is achieved by introducing a predicate $is_{boolean}$ that is used to declare that a variable is boolean, and by transforming predicates on booleans into predicates over integers such that $P(n) \Leftrightarrow P(n')$ for every $n, n' \neq 0$.

Third, most, if not all, Java compilers do perform program optimizations. Dealing with advanced program optimizations is out of scope of this paper, but there are some very simple optimizations that are performed by many compilers. Some of this simple optimizations are transforming $\mathsf{if}(\mathsf{true})\{s_t\}\{s_f\}$ into s_t, and $\mathsf{while}(\mathsf{false})\{s\}$ into skip. These optimizations break preservation of proof obligations. Nevertheless, one can preserve provability of specifications, and transform certificates of source programs into certificates of bytecode programs without much effort. However, for these examples of optimizations, it makes more sense to perform a source-to-source transformation of programs before proving properties about them. For more advanced optimizations, the reader is referred to the related work in Section 2.

7 Conclusion

Preservation of proof obligations establishes a strong relationship between verification of source programs and verification of the corresponding compiled programs, and from the perspective of PCC, warrants the use of source code verification to generate certificates of complex properties. In this article, we show that preservation of proof obligations holds for an idealization of the Java framework. In parallel, Burdy and Pavlova (see [16]) have developed a prototype implementation of a proof-transforming compiler for Java, using slightly different verification condition generation mechanisms. Within the context of the Mobius project[1], Charles, Grégoire and Lehner are developing a proof-transforming compiler that targets a verification condition generator that has been fully formalized and verified in Coq. The compiler relies on the tactic language of Coq to overcome the aforementioned difficulties with naming, booleans, and optimizations.

However, the verification condition generators considered in this paper do not scale well, in particular because of the explosion of the control flow graph that arises in object-oriented programs. In order to curb this explosion, we are developing hybrid certificates, which provide typing information that can be verified automatically and that is used by the verification condition generator to generate more compact proof obligations [8]. Our next objective is to extend preservation of proof obligations to such hybrid certificates.

References

1. Balakrishnan, G.: WYSISWYX: What you see is not what you execute. PhD thesis, Department of Computer Science, University of Wisconsin (2007)
2. Bannwart, F.Y., Müller, P.: A program logic for bytecode. In: Spoto, F. (ed.) Bytecode Semantics, Verification, Analysis and Transformation. Electronic Notes in Theoretical Computer Science, vol. 141, pp. 255–273. Elsevier, Amsterdam (2005)
3. Barthe, G., Grégoire, B., Kunz, C., Rezk, T.: Certificate translation for optimizing compilers. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 301–317. Springer, Heidelberg (2006)
4. Barthe, G., Kunz, C.: Certificate translation in abstract interpretation. In: Drossopoulou, S. (ed.) European Symposium on Programming, Budapest, Hungary. LNCS, vol. 4960, pp. 368–382. Springer, Heidelberg (2008)

[1] See `http://mobius.inria.fr`.

5. Barthe, G., Rezk, T., Saabas, A.: Proof obligations preserving compilation. In: Dimitrakos, T., Martinelli, F., Ryan, P.Y.A., Schneider, S. (eds.) FAST 2005. LNCS, vol. 3866, pp. 112–126. Springer, Heidelberg (2006)
6. Burdy, L., Pavlova, M.: Java bytecode specification and verification. In: Symposium on Applied Computing, pp. 1835–1839. ACM Press, New York (2006)
7. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Principles of Programming Languages, pp. 238–252 (1977)
8. Grégoire, B., Sacchini, J.: Combining a verification condition generator for a bytecode language with static analyses. In: Barthe, G., Fournet, C. (eds.) Trustworthy Global Computing. LNCS, vol. 4912, pp. 23–40. Springer, Heidelberg (2007)
9. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: Morrisett, J.G., Peyton Jones, S.L. (eds.) Principles of Programming Languages, pp. 42–54. ACM Press, New York (2006)
10. Logozzo, F., Fähndrich, M.: On the relative completeness of bytecode analysis versus source code analysis. In: Hendren, L. (ed.) CC. LNCS, vol. 4959, pp. 197–212. Springer, Heidelberg (2008)
11. Müller, P., Nordio, M.: Proof-transforming compilation of programs with abrupt termination. In: SAVCBS 2007: Proceedings of the 2007 conference on Specification and verification of component-based systems, pp. 39–46. ACM Press, New York (2007)
12. Necula, G.C.: Proof-carrying code. In: Principles of Programming Languages, pp. 106–119. ACM Press, New York (1997)
13. Necula, G.C., Lee, P.: The design and implementation of a certifying compiler. In: Programming Languages Design and Implementation, vol. 33, pp. 333–344. ACM Press, New York (1998)
14. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL. A Proof Assistant for Higher-Order Logic. Springer, Heidelberg (2002)
15. Nordio, M., Müller, P., Meyer, B.: Formalizing proof-transforming compilation of eiffel programs. Technical Report 587, ETH Zurich (2008)
16. Pavlova, M.: Specification and verification of Java bytecode. PhD thesis, Université de Nice Sophia-Antipolis (2007)
17. Rival, X.: Symbolic Transfer Functions-based Approaches to Certified Compilation. In: Principles of Programming Languages, pp. 1–13. ACM Press, New York (2004)
18. Saabas, A., Uustalu, T.: A compositional natural semantics and Hoare logic for low-level languages. Theoretical Computer Science 373(3), 273–302 (2007)
19. Saabas, A., Uustalu, T.: Proof optimization for partial redundancy elimination. In: ACM Workshop on Partial Evaluation and Semantics-based Program Manipulation, pp. 91–101. ACM Press, New York (2008)

Efficient Well-Definedness Checking

Ádám Darvas, Farhad Mehta, and Arsenii Rudich

ETH Zurich, Switzerland
{adam.darvas,farhad.mehta,arsenii.rudich}@inf.ethz.ch

Abstract. Formal specifications often contain partial functions that may lead to ill-defined terms. A common technique to eliminate ill-defined terms is to require well-definedness conditions to be proven. The main advantage of this technique is that it allows us to reason in a two-valued logic even if the underlying specification language has a three-valued semantics. Current approaches generate well-definedness conditions that grow exponentially with respect to the input formula. As a result, many tools prove shorter, but stronger approximations of these well-definedness conditions instead.

We present a procedure which generates well-definedness conditions that grow linearly with respect to the input formula. The procedure has been implemented in the Spec# verification tool. We also present empirical results that demonstrate the improvements made.

1 Introduction

Formal specifications often allow terms to contain applications of partial functions, such as division $x\,/\,y$ or factorial $\mathit{fact}(z)$. However, it is not clear what value $x\,/\,y$ yields if y is 0, or what value $\mathit{fact}(z)$ yields if z is negative. Specification languages need to handle *ill-defined terms*, that is, either have to define the semantics of partial-function applications whose arguments fall outside their domains or have to eliminate such applications.

One of the standard approaches to handle ill-defined terms is to define a three-valued semantics [22] by considering ill-defined terms to have a special value, *undefined*, denoted by $\perp$. That is, both $x\,/\,0$ and $\mathit{fact}(-5)$ are considered to evaluate to $\perp$. In order to reason about specifications with a three-valued semantics, undefinedness is lifted to formulas by extending their denoted truth values to $\{\mathbf{true}, \mathbf{false}, \perp\}$.

A common technique to reason about specifications with a three-valued semantics is to *eliminate* ill-defined terms before starting the actual proof. *Well-definedness conditions* are generated, whose validity ensures that all formulas at hand can be evaluated to either **true** or **false**. That is, once the well-definedness conditions have been discharged, $\perp$ is guaranteed to never be encountered.

The advantage of the technique is that both the well-definedness conditions and the actual proof obligations are to be proven in classical two-valued logic, which is simpler, better understood, more widely used, and has better automated tool support [30] than three-valued logics.

The technique of eliminating ill-defined terms in specifications by generating well-definedness conditions is used in several approaches, for instance, B [2], PVS [13], CVC Lite [6], and ESC/Java2 [21].

Motivation. A drawback of this approach is that well-definedness conditions can be very large, causing significant time overhead in the proof process. As an example, consider the following formula:

$$x\,/\,y = c_1 \ \wedge \ \mathit{fact}(y) = c_2 \ \wedge \ y > 5 \tag{1}$$

where x and y are variables, and c_1 and c_2 are constants. The formula is well-defined, that is, it always evaluates to either **true** or **false**. This can be justified by a case split on the third conjunct, which is always well-defined:

1. if the third conjunct evaluates to **true**, then the division and factorial functions are known to be applied within their domains, and thus, the first and second conjuncts can be evaluated to **true** or **false**. This means that the whole formula can be evaluated to either **true** or **false**.
2. if the third conjunct evaluates to **false**, then the whole formula evaluates to **false** (according to the semantics we use in the paper).

The literature [8,27,3,9] proposes the procedure $\mathcal{D}$ to generate well-definedness conditions. The procedure is *complete* [8,9], that is, the well-definedness condition generated from a formula is provable if and only if the formula is well-defined. Procedure $\mathcal{D}$ would generate the following condition for (1):

$$\begin{aligned}
&(y \neq 0 \wedge (y \geq 0 \vee (y \geq 0 \wedge \mathit{fact}(y) \neq c_2) \vee y \leq 5)) \ \vee \\
&(y \neq 0 \wedge x/y \neq c_1) \ \vee \\
&((y \geq 0 \vee (y \geq 0 \wedge \mathit{fact}(y) \neq c_2) \vee y \leq 5) \wedge \neg(\mathit{fact}(y) \neq c_2 \wedge y > 5))
\end{aligned}$$

As expected, the condition is provable. However, the size of the condition is striking, given that the original formula contained only three sub-formulas and two partial-function applications. In fact, procedure $\mathcal{D}$ yields well-formedness conditions that *grow exponentially* with respect to the input formula. This is a major problem for tools that have to prove well-definedness of considerably larger formulas than (1), for instance, the well-definedness conditions for B models, as presented in [8].

Due to the exponential blow-up of well-definedness conditions, the $\mathcal{D}$ procedure is not used in practice [8,27,3]. Instead, another procedure $\mathcal{L}$ is used, which generates much smaller conditions with linear growth, but which is *incomplete.* That is, the procedure may generate unprovable well-definedness conditions for well-defined formulas. This is the case with formula (1), for which the procedure would yield the following unprovable condition:

$$y \neq 0 \wedge (x/y = c_1 \Rightarrow y \geq 0)$$

Incompleteness of the procedure originates from its "sensitivity" to the order of sub-formulas. For instance, after proper re-ordering of the sub-formulas of

([1]), the procedure would yield a provable condition. This may be tedious for large formulas and may appear unnatural to users who are familiar with logics in which the order of sub-formulas is irrelevant for proof. Furthermore, there are situations (for instance, our example in Section 4) where such a manual re-ordering cannot be done.

Contributions. Our main contribution is a new procedure $\mathcal{Y}$, which unifies the advantages of $\mathcal{D}$ and $\mathcal{L}$, while eliminating their weaknesses. That is, (1) $\mathcal{Y}$ yields well-definedness conditions that *grow linearly* with respect to the size of the input formula, and (2) $\mathcal{Y}$ is equivalent to $\mathcal{D}$, and therefore complete and insensitive to the order of sub-formulas. To our knowledge, this is the first procedure that has *both* of these two properties.

The definition of the new procedure is very intuitive and straightforward. We prove that it is equivalent with $\mathcal{D}$ in two ways: (1) in a syntactical manner, as $\mathcal{D}$ was derived in [3], and (2) in a semantical way, as $\mathcal{D}$ was introduced in [8].

We have implemented the new procedure in the Spec# verification tool [5] in the context of the well-formedness checking of method specifications [26]. We have compared our procedure with $\mathcal{D}$ and $\mathcal{L}$ using two automated theorem provers. The empirical results clearly show that not only the size of generated well-definedness conditions are significantly smaller than what $\mathcal{D}$ produces, but the time to prove validity of the conditions is also decreased by the use of $\mathcal{Y}$. Furthermore, our results show that the performance of $\mathcal{Y}$ is also better than that of $\mathcal{L}$ in terms of the size of generated conditions.

Outline. The rest of the paper is structured as follows. Section 2 formally defines procedures $\mathcal{D}$ and $\mathcal{L}$, and highlights their main differences. Section 3 presents our main contribution: the $\mathcal{Y}$ procedure and the proof of its equivalence with $\mathcal{D}$. Section 4 demonstrates the improvements of our approach through empirical results. We discuss related work in Section 5 and conclude in Section 6.

2 Eliminating Ill-definedness

The main idea behind the technique of eliminating ill-definedness in specifications is to reduce the three-valued domain to a two-valued domain by ensuring that $\perp$ is never encountered. $\mathcal{D}$ is used for this purpose. Hoogewijs introduced $\mathcal{D}$ in the form of the logical connective Δ in [19], and proposed a first-order calculus, which includes this connective. Later, $\mathcal{D}$ was reformulated as a formula transformer, for instance, in [8,3,9] for the above syntax. $\mathcal{D}$ takes a formula ϕ and produces another formula $\mathcal{D}(\phi)$. The interpretation of the formula $\mathcal{D}(\phi)$ in two-valued logic is **true** if and only if the interpretation of ϕ in three-valued logic is different from $\perp$.

In order to have a basis for formal definitions, we define the syntax of terms and formulas that we consider in this paper. We follow the standard syntax-definition given in Figure 1. Throughout the paper we use **true**, **false**, and $\perp$ to denote the semantic truth values, and *true* and *false* to refer to the syntactic entities.

$$\begin{array}{llll} \mathit{Term} ::= \mathit{Var} & \qquad \mathit{Formula} ::= P(t_1, \ldots, t_n) & & \\ \quad\ \mid\ f(t_1, \ldots, f_n) & \qquad\qquad \mid\ \mathit{true} & \mid\ \mathit{false} & \\ & \qquad\qquad \mid\ \neg\phi & & \\ & \qquad\qquad \mid\ \phi_1 \wedge \phi_2 & \mid\ \phi_1 \vee \phi_2 & \\ & \qquad\qquad \mid\ \forall x.\ \phi & \mid\ \exists x.\ \phi & \end{array}$$

Fig. 1. Syntax of terms and formulas we consider in this paper

$$\begin{array}{lll}
\delta(\mathit{Var}) & \triangleq & \mathit{true} \\
\delta(f(e_1, \ldots, e_n)) & \triangleq & d_f(e_1, \ldots, e_n) \ \wedge \ \bigwedge_{i=1}^{n} \delta(e_i) \\
\mathcal{D}(P(e_1, \ldots, e_n)) & \triangleq & \bigwedge_{i=1}^{n} \delta(e_i) \\
\mathcal{D}(\mathit{true}) & \triangleq & \mathit{true} \\
\mathcal{D}(\mathit{false}) & \triangleq & \mathit{true} \\
\mathcal{D}(\neg\phi) & \triangleq & \mathcal{D}(\phi) \\
\mathcal{D}(\phi_1 \wedge \phi_2) & \triangleq & (\mathcal{D}(\phi_1) \wedge \mathcal{D}(\phi_2)) \ \vee \ (\mathcal{D}(\phi_1) \wedge \neg\phi_1) \ \vee \ (\mathcal{D}(\phi_2) \wedge \neg\phi_2) \\
\mathcal{D}(\phi_1 \vee \phi_2) & \triangleq & (\mathcal{D}(\phi_1) \wedge \mathcal{D}(\phi_2)) \ \vee \ (\mathcal{D}(\phi_1) \wedge \phi_1) \ \vee \ (\mathcal{D}(\phi_2) \wedge \phi_2) \\
\mathcal{D}(\forall x.\ \phi) & \triangleq & \forall x. \mathcal{D}(\phi) \ \vee \ \exists x. (\mathcal{D}(\phi) \wedge \neg\phi) \\
\mathcal{D}(\exists x.\ \phi) & \triangleq & \forall x. \mathcal{D}(\phi) \ \vee \ \exists x. (\mathcal{D}(\phi) \wedge \phi)
\end{array}$$

Fig. 2. Definition of the δ and $\mathcal{D}$ operators as given by Behm et al. [8]

2.1 Defining the $\mathcal{D}$ Operator

The definition of $\mathcal{D}$ is given in Figure 2. Operator δ handles terms and $\mathcal{D}$ handles formulas. A variable is always well-defined. Application of function f is well-defined if and only if f's *domain restriction* d_f holds and all parameters e_i are well-defined. Each function is associated with a domain restriction, which is a predicate that represents the domain of the function. Such predicates should only contain total-function applications. For instance, the domain restriction of the factorial function is that the parameter is non-negative.

A predicate is well-defined if and only if all parameters are well-defined. Note that this definition assumes predicates to be total. Although an extension to partial predicates is straightforward, we use this definition for simplicity and to

$\wedge$	**true**	**false**	$\perp$
true	**true**	**false**	$\perp$
false	**false**	**false**	**false**
$\perp$	$\perp$	**false**	$\perp$

(*a*) Strong Kleene

$\wedge$	**true**	**false**	$\perp$
true	**true**	**false**	$\perp$
false	**false**	**false**	**false**
$\perp$	$\perp$	$\perp$	$\perp$

(*b*) McCarthy

Fig. 3. Kleene's and McCarthy's interpretation of conjunction

have a direct comparison of our approach to [8]. Constants *true* and *false* are always well-defined. Well-definedness of logical connectives is defined according to Strong Kleene connectives [22]. For instance, as the truth table in Figure 3(a) shows, a conjunction is well-defined if and only if either (1) both conjuncts are well-defined, or (2) if one of the conjuncts is well-defined and evaluates to **false**. Intuitively, in case (1) the classical two-valued evaluation can be applied, while in case (2) the truth value of the conjunction is **false** independently of the well-definedness and value of the other conjunct.

Well-definedness of universal quantification can be thought of as the generalization of the well-definedness of conjunction. Disjunction and existential quantification are the duals of conjunction and universal quantification, respectively. Soundness and completeness of $\mathcal{D}$ was proven in [19,8,9].

2.2 An Approximation of the $\mathcal{D}$ Operator

As mentioned before in Section 1, the problem with the $\mathcal{D}$ operator is that it yields well-definedness conditions that grow exponentially with respect to the size of the input formula. This problem has been recognized in several approaches, for instance, in B [3] and PVS [27]. As a consequence, these approaches use a simpler, but stricter operator $\mathcal{L}$ [8,3] for computing well-definedness conditions. The definition of $\mathcal{L}$ differs from that of $\mathcal{D}$ only for the following connectives:[1]

$$\begin{array}{lcllcl} \mathcal{L}(\phi_1 \wedge \phi_2) & \triangleq & \mathcal{L}(\phi_1) \wedge (\phi_1 \Rightarrow \mathcal{L}(\phi_2)) & \quad \mathcal{L}(\forall x.\ \phi) & \triangleq & \forall x. \mathcal{L}(\phi) \\ \mathcal{L}(\phi_1 \vee \phi_2) & \triangleq & \mathcal{L}(\phi_1) \wedge (\neg\phi_1 \Rightarrow \mathcal{L}(\phi_2)) & \quad \mathcal{L}(\exists x.\ \phi) & \triangleq & \forall x. \mathcal{L}(\phi) \end{array}$$

Looking at the definition, we can see that $\mathcal{L}$ yields well-definedness conditions that grow linearly with respect to the input formula. This is a great advantage over $\mathcal{D}$. However, the $\mathcal{L}$ operator is stronger than $\mathcal{D}$, that is, $\mathcal{L}(\phi) \Rightarrow \mathcal{D}(\phi)$ holds, but $\mathcal{D}(\phi) \Rightarrow \mathcal{L}(\phi)$ does not necessarily hold, as shown for formula (1) in Section 1. This means that we lose completeness with the use of $\mathcal{L}$.

For quantifiers, $\mathcal{L}$ requires that the quantified formula is well-defined for all instantiations of the quantified variable. As a result, a universal quantification may be considered ill-defined although an instance is known to evaluate to **false**. Similarly, an existential quantification may also be considered ill-defined although an instance is known to evaluate to **true**. The $\mathcal{D}$ operator takes these "short-circuits" into account.

The other source of incompleteness originates from defining conjunction and disjunction according to McCarthy's interpretation [24], which evaluates formulas *sequentially*. That is, if the first operand of a connective is $\perp$, then the result is defined to be $\perp$, independently of the second operand. The truth table in Figure 3(b) presents McCarthy's interpretation of conjunction. The only difference from Kleene's interpretation is in the interpretation of $\perp \wedge$ **false**, which yields $\perp$. This reveals the most important difference between the two interpretations: in McCarthy's interpretation conjunction and disjunction are not commutative.

[1] Although our formula-syntax does not contain implication, we use it below to keep the intuition behind the definition.

As a consequence, for instance, $\phi_1 \wedge \phi_2$ may be considered ill-defined, although $\phi_2 \wedge \phi_1$ is considered well-defined. Such cases might come unexpected to users who are used to classical logic where conjunction and disjunction are commutative.

In most cases this incompleteness issue can be resolved by manually re-ordering sub-formulas. However, as pointed out by Rushby et al. [27], the manual re-ordering of sub-formulas is not an option when specifications are automatically generated from some other representation. Furthermore, Cheng and Jones [12], and Rushby et al. [27] give examples for which even manual re-ordering does not help, and well-defined formulas are inevitably rejected by $\mathcal{L}$.

3 An Efficient Equivalent of the $\mathcal{D}$ Operator

In this section we present our main contribution: a new procedure $\mathcal{Y}$ that yields considerably smaller well-definedness conditions than $\mathcal{D}$, and that retains completeness. We prove equivalence of $\mathcal{Y}$ and $\mathcal{D}$ in two ways: (1) we syntactically derive the definition of $\mathcal{Y}$, (2) using three-valued interpretation we prove by induction that the definition of $\mathcal{Y}$ is equivalent to that of $\mathcal{D}$. Both proofs demonstrate the intuitive and simple nature of $\mathcal{Y}$'s definition.

3.1 Syntactical Derivation of $\mathcal{Y}$

We introduce two new formula transformers $\mathcal{T}$ and $\mathcal{F}$, and define them as follows:

$$\mathcal{T}(\phi) \triangleq \mathcal{D}(\phi) \wedge \phi \qquad \text{and} \qquad \mathcal{F}(\phi) \triangleq \mathcal{D}(\phi) \wedge \neg\,\phi$$

That is, $\mathcal{T}(\phi)$ yields **true** if and only if ϕ is well-defined and evaluates to **true**. Analogously, $\mathcal{F}(\phi)$ yields **true** if and only if ϕ is well-defined and evaluates to **false**. From the definitions the following theorem follows.

Theorem 1. $\mathcal{D}(\phi) \Leftrightarrow \mathcal{T}(\phi) \vee \mathcal{F}(\phi)$
Proof. $\mathcal{D}(\phi) \Leftrightarrow \mathcal{D}(\phi) \wedge (\phi \vee \neg\,\phi) \Leftrightarrow (\mathcal{D}(\phi) \wedge \phi) \vee (\mathcal{D}(\phi) \wedge \neg\,\phi) \Leftrightarrow \mathcal{T}(\phi) \vee \mathcal{F}(\phi)$ □

Intuitively, the theorem expresses that formula ϕ is well-defined if and only if ϕ evaluates either to **true** or to **false**. This directly corresponds to the interpretation of $\mathcal{D}$ given by Hoogewijs [19].

From the definitions of $\mathcal{T}$ and $\mathcal{F}$, the equivalences presented in Figure 4 can be derived. To demonstrate the simplicity of these derivations, we give the derivation of $\mathcal{T}(\phi_1 \wedge \phi_2)$ and $\mathcal{F}(\forall\, x.\ \phi)$:

$$\begin{aligned}
&\mathcal{T}(\phi_1 \wedge \phi_2) \Leftrightarrow \mathcal{D}(\phi_1 \wedge \phi_2) \wedge \phi_1 \wedge \phi_2 \Leftrightarrow \\
&((\mathcal{D}(\phi_1) \wedge \mathcal{D}(\phi_2)) \vee (\mathcal{D}(\phi_1) \wedge \neg\phi_1) \vee (\mathcal{D}(\phi_2) \wedge \neg\phi_2)) \wedge \phi_1 \wedge \phi_2 \Leftrightarrow \\
&\mathcal{D}(\phi_1) \wedge \mathcal{D}(\phi_2) \wedge \phi_1 \wedge \phi_2 \Leftrightarrow \mathcal{T}(\phi_1) \wedge \mathcal{T}(\phi_2)
\end{aligned}$$

$$\mathcal{T}(P(e_1, .., e_n)) \Leftrightarrow P(e_1, .., e_n) \wedge \bigwedge_{i=1}^{n} \delta(e_i) \quad \mathcal{F}(P(e_1, .., e_n)) \Leftrightarrow \neg P(e_1, .., e_n) \wedge \bigwedge_{i=1}^{n} \delta(e_i)$$

$$\begin{array}{lcl@{\qquad}lcl}
\mathcal{T}(\mathit{true}) & \Leftrightarrow & \mathit{true} & \mathcal{F}(\mathit{true}) & \Leftrightarrow & \mathit{false} \\
\mathcal{T}(\mathit{false}) & \Leftrightarrow & \mathit{false} & \mathcal{F}(\mathit{false}) & \Leftrightarrow & \mathit{true} \\
\mathcal{T}(\neg\phi) & \Leftrightarrow & \mathcal{F}(\phi) & \mathcal{F}(\neg\phi) & \Leftrightarrow & \mathcal{T}(\phi) \\
\mathcal{T}(\phi_1 \wedge \phi_2) & \Leftrightarrow & \mathcal{T}(\phi_1) \wedge \mathcal{T}(\phi_2) & \mathcal{F}(\phi_1 \wedge \phi_2) & \Leftrightarrow & \mathcal{F}(\phi_1) \vee \mathcal{F}(\phi_2) \\
\mathcal{T}(\phi_1 \vee \phi_2) & \Leftrightarrow & \mathcal{T}(\phi_1) \vee \mathcal{T}(\phi_2) & \mathcal{F}(\phi_1 \vee \phi_2) & \Leftrightarrow & \mathcal{F}(\phi_1) \wedge \mathcal{F}(\phi_2) \\
\mathcal{T}(\forall x.\ \phi) & \Leftrightarrow & \forall x.\ \mathcal{T}(\phi) & \mathcal{F}(\forall x.\ \phi) & \Leftrightarrow & \exists x.\ \mathcal{F}(\phi) \\
\mathcal{T}(\exists x.\ \phi) & \Leftrightarrow & \exists x.\ \mathcal{T}(\phi) & \mathcal{F}(\exists x.\ \phi) & \Leftrightarrow & \forall x.\ \mathcal{F}(\phi)
\end{array}$$

Fig. 4. Derived equivalences for $\mathcal{T}$ and $\mathcal{F}$

$$\begin{array}{l}
\mathcal{F}(\forall x.\ \phi) \ \Leftrightarrow\ \mathcal{D}(\forall x.\ \phi) \wedge \neg\forall x.\ \phi \ \Leftrightarrow \\
(\forall x.\ \mathcal{D}(\phi) \vee (\exists x.\ (\mathcal{D}(\phi) \wedge \neg\phi))) \wedge \exists x.\ \neg\phi \ \Leftrightarrow \\
\exists x.\ (\mathcal{D}(\phi) \wedge \neg\phi) \ \Leftrightarrow\ \exists x.\ \mathcal{F}(\phi)
\end{array}$$

The derived equivalences are very intuitive. Both $\mathcal{T}$ and $\mathcal{F}$ reflect the standard two-valued interpretation of formulas. For instance, $\mathcal{F}$ essentially realizes de Morgan's laws. The handling of terms is the same as before using the δ operator. Note that $\mathcal{T}$ and $\mathcal{F}$ are mutually recursive in the equivalences. Termination of the mutual application of the operators is trivially guaranteed: the size of formulas yields the measure for termination.

The more involved semantic proof of equivalence is presented in the appendix. The proof, in particular **Lemma 4**, highlights the intuition behind $\mathcal{Y}$'s definition.

The definition of our new procedure $\mathcal{Y}$, based on **Theorem 1**, is the following:

$$\mathcal{Y}(\phi) \ \triangleq\ \mathcal{T}(\phi) \ \vee\ \mathcal{F}(\phi)$$

It is easy to see that (1) our procedure begins by duplicating the size of the input formula ϕ, and (2) afterwards applies operators $\mathcal{T}$ and $\mathcal{F}$ that yield formulas that are linear in size with respect to their input formulas.

That is, overall our procedure yields well-definedness conditions that grow linearly with respect to the size of the input formula. This is a significant improvement over $\mathcal{D}$ which yields formulas that are exponential in size with respect to the input formula. Intuitively, this improvement can be explained as follows: $\mathcal{D}$ makes case distinctions on the well-definedness of sub-formulas at each step of its application, whereas $\mathcal{Y}$ only performs a single initial case distinction on the validity of the entire formula. In spite of this difference, our procedure is equivalent to $\mathcal{D}$, thus it is symmetric, as opposed to $\mathcal{L}$.

4 Implementation and Empirical Results

We have implemented a well-formedness checker in the context of the verification of object-oriented programs. Our implementation extends the Spec# verification tool [5] by a new module that performs well-definedness and well-foundedness checks on specifications using the $\mathcal{Y}$ procedure. Details of the technique applied in the well-formedness checker are described in [26].

Additionally, in order to be able to compare the different procedures, we have built a prototype that implements the $\mathcal{D}$, $\mathcal{L}$, and $\mathcal{Y}$ procedures for the syntax given in Figure 1. We used the two automated theorem provers that are integrated with Spec#: Simplify [15] and Z3 [14], both of which are used by several other tools as prover back-ends too. The experiment was performed on a machine with Intel Pentium M (1.86 GHz) and 2 GB RAM.

The benchmark. We have used the following inductively defined formula, which allowed us to experiment with formula sizes that grow linearly with respect to n, and which is well-defined for every natural number n:

$$\begin{array}{lcl} \phi_0 & \triangleq & f(x) = x \ \vee \ f(-x) = -x \\ \phi_n & \triangleq & \phi_{n-1} \ \wedge \ (f(x+n) = x+n \ \vee \ f(-x-n) = -x-n) \end{array}$$

where the definition and domain restriction of f is as follows:

$$\forall x.\ x \geq 0 \ \Rightarrow \ f(x) = x \qquad \text{and} \qquad d_f\!:\ x \geq 0$$

Note that formula ϕ_n is well-defined for any n. However, its well-definedness cannot be proven using $\mathcal{L}$ for any n, and no re-ordering would help this situation.

Empirical results. Figure 5(a) shows that well-definedness conditions generated by $\mathcal{D}$ grow exponentially, whereas conditions generated by $\mathcal{L}$ and $\mathcal{Y}$ grow linearly. This was expected from their definitions. Note that the y axis uses a logarithmic scale. The figure also shows, that the sizes of conditions generated using $\mathcal{Y}$ are smaller than those generated by $\mathcal{L}$ for $n > 4$.

Figure 5(b) compares the time that Simplify (version 1.5.4) required to prove the well-definedness conditions generated from our input formula. As required by its interface, these conditions were given to Simplify as plain text. We see that the time required to prove formulas generated by $\mathcal{D}$ grows exponentially, whereas with $\mathcal{Y}$ the required time grows linearly. Note that the y axis uses a logarithmic scale. Additionally, for $\mathcal{D}$ our prototype was not able to generate the well-definedness condition for input formulas with $n > 16$ because it ran out of memory.

Figure 5(c) shows the results of the same experiment using Z3 (version 1.2). Note that the y axis is linear. From this graph we see that although the times required to prove well-definedness conditions show the same growth pattern for both procedure $\mathcal{D}$ and $\mathcal{Y}$, the times recorded for $\mathcal{Y}$ are approximately 1/3 to 1/2 below that of for $\mathcal{D}$. For instance, with $n = 200$, Z3 proves the condition generated by $\mathcal{D}$ in 9 seconds, while it takes 3.5 seconds for the condition generated by $\mathcal{Y}$. For $n = 300$, these figures are 23.5 and 10.5 seconds, respectively. Note that we could successfully prove much larger well-definedness conditions generated by $\mathcal{D}$ in Z3 as compared to Simplify. This is because (1) we used the native API of Z3 in order to construct formulas with maximal sharing, and (2) due to the

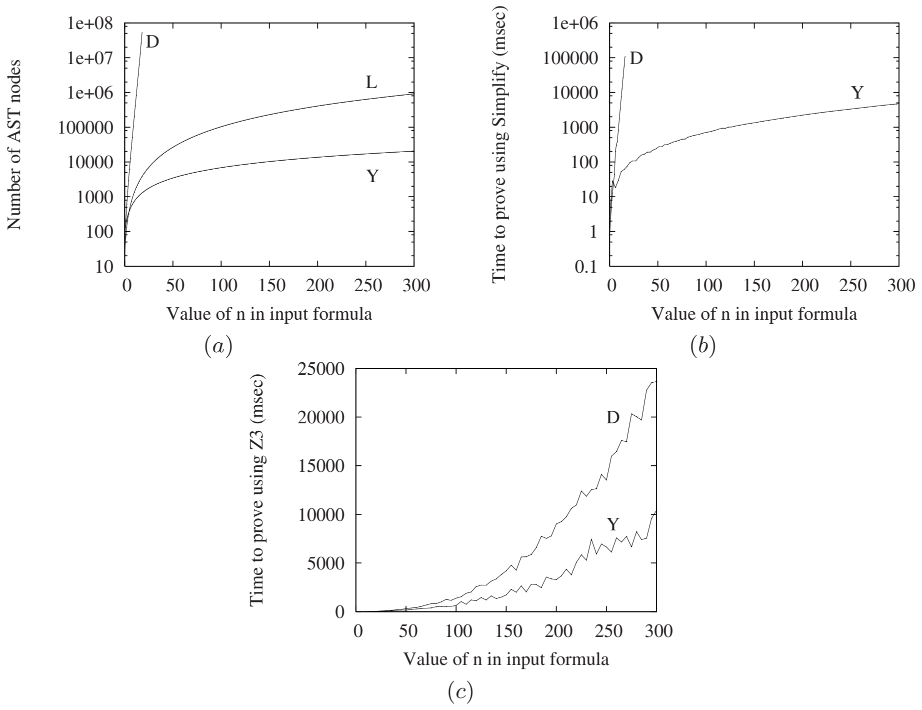


Fig. 5. (*a*) Size of well-definedness conditions generated by procedures $\mathcal{D}$, $\mathcal{L}$, and $\mathcal{Y}$; (*b*) Time to prove well-definedness conditions using Simplify; (*c*) Time to prove well-definedness conditions using Z3

use of its API, Z3 may have benefited from sub-formula sharing, which could have made the size of the resulting formula representation linear. In spite of this, procedure $\mathcal{Y}$ performs better than $\mathcal{D}$.

Note that Figure 5(*b*) and 5(*c*) do not plot the results of $\mathcal{L}$. This is because the $\mathcal{L}$ procedure cannot prove well-definedness of the input formulas.

Finally, we note that the whole sequence of formulas were passed to a single session of Simplify or Z3, respectively.

5 Related Work

The handling of partial functions in formal specifications has been studied extensively and several different approaches have been proposed. Here we only mention three mainstream approaches and refer the reader for detailed accounts to Arthan's paper [4], which classifies different approaches to undefinedness, to Abrial and Mussat's paper [3, Section 1.7], and to Hähnle's survey [18].

Eliminating Undefinedness. As mentioned already in the paper, eliminating undefinedness by the generation of well-definedness conditions is a common technique to handle partial functions, and is applied in several approaches, such as B [8,3], PVS [27], CVC Lite [9], and ESC/Java2 [11]. The two procedures proposed in these papers are $\mathcal{D}$ and $\mathcal{L}$.

PVS combines proving well-definedness conditions with type checking. In PVS, partial functions are modeled as total functions whose domain is a predicate subtype [27]. This makes the type system undecidable requiring Type Correctness Conditions to be proven. PVS uses the $\mathcal{L}$ operator because $\mathcal{D}$ was found to be inefficient [27].

CVC Lite uses the $\mathcal{D}$ procedure for the well-definedness checking of formulas. Berezin et al. [9] mention that if formulas are represented as DAGs, then the worst-case size of $\mathcal{D}(\phi)$ is linear with respect to the size of ϕ. However, there are no empirical results presented to confirm any advantages of using the DAG representation in terms of proving times.

Recent work on ESC/Java2 by Chalin [11] requires proving the well-definedness of specifications written in the Java Modeling Language (JML) [23]. Chalin uses the $\mathcal{L}$ procedure, however, as opposed to other approaches, not because of inefficiency issues. The $\mathcal{L}$ procedure directly captures the semantics of conditional boolean operators (e.g. `&&` and `||` in Java) that many programming languages contain, and which are often used, for instance, in JML specifications. Chalin's survey [10] indicates that the use of $\mathcal{L}$ is better suited for program verification than $\mathcal{D}$, since it yields well-definedness conditions that are closer to the expectations of programmers.

Schieder and Broy [28] propose a different approach to the checking of well-definedness of formulas. They define a formula under a three-valued interpretation to be well-defined if and only if its interpretation yields **true** both if $\perp$ is interpreted as **true**, and if $\perp$ is interpreted as **false**. Although checking well-definedness of formulas becomes relatively simple, the interpretation may be unintuitive for users. For example, formula $\perp \vee \neg\perp$ is considered to be well-defined. We prefer to eliminate such formulas by using classical Kleene logic.

Three-Valued Logics. Another standard way to handle partial functions is to fully integrate ill-defined terms into the formal logic by developing a three-valued logic. This approach is attributed to Kleene [22]. A well-known three-valued logic is LPF [7,12] developed by C.B. Jones et al. in the context of VDM [20]. Other languages that follow this approach include Z [29] and OCL [1].

A well-known drawback of three-valued logics is that they may seem unnatural to proof engineers. For instance, in LPF, the law of the excluded middle and the deduction rule (a.k.a. ImpI) do not hold. Furthermore, a second notion of equality (called "weak equality") is required to avoid proving, for instance, that $x \,/\, 0 = \mathit{fact}(-5)$ holds. Another major drawback is that there is significantly less tool support for three-valued logics than there is for two-valued logics.

Underspecification. The approach of underspecification assigns an ill-defined term a definite, but unknown value from the type of the term [16]. Thus, the resulting interpretation is two-valued, however, in certain cases the truth value of formulas cannot be determined due to the unknown values. For instance, the truth value of $x / 0 = fact(-5)$ is known to be either **true** or **false**, but there is no way to deduce which of the two. However, for instance, $x / 0 = x / 0$ is trivially provable. This might not be a desired behavior. For instance, the survey by Chalin [10] argues that this is against the intuition of programmers, who would rather expect an error to occur in the above case. Underspecification is applied, for instance, in the Isabelle theorem prover [25], the Larch specification language [17], and JML [23].

6 Conclusion

A commonly applied technique to handle partial-function applications in formal specifications is to pose well-definedness conditions, which guarantee that undefined terms and formulas are never encountered. This technique allows one to use two-valued logic to reason about specifications that have a three-valued semantics. Previous work proposed two procedures, each having some drawback. The $\mathcal{D}$ procedure yields formulas that are too large to be used in practice. The $\mathcal{L}$ procedure is incomplete, resulting in the rejection of well-defined formulas.

In this paper we proposed a new procedure $\mathcal{Y}$, which eliminates these drawbacks: $\mathcal{Y}$ is complete and yields formulas that grow linearly with respect to the size of the input formula. Approaches that apply the $\mathcal{D}$ or $\mathcal{L}$ procedures (for instance, B, PVS, and CVC Lite) could benefit from our procedure. The required implementation overhead would be minimal.

Our procedure has been implemented in the Spec# verification tool to enforce well-formedness of invariants and method specifications. Additionally, we implemented a prototype to allow us to compare the new procedure with $\mathcal{D}$ and $\mathcal{L}$. Beyond the expected benefits of shorter well-definedness conditions, our experiments also show that theorem provers need less time to prove the conditions generated using $\mathcal{Y}$.

Acknowledgments. We would like to thank Peter Müller and the anonymous reviewers for their helpful comments. This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project.

References

1. UML 2.0 OCL Specification (May 2006), `http://www.omg.org/docs/formal/06-05-01.pdf`
2. Abrial, J.-R.: The B Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)
3. Abrial, J.-R., Mussat, L.: On using conditional definitions in formal theories. In: Bert, D., Bowen, J.P., Henson, M.C., Robinson, K. (eds.) B 2002 and ZB 2002. LNCS, vol. 2272, pp. 242–269. Springer, Heidelberg (2002)
4. Arthan, R.: Undefinedness in Z: Issues for specification and proof. In: CADE Workshop on Mechanization of Partial Functions (1996)

5. 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, pp. 49–69. Springer, Heidelberg (2005)
6. Barrett, C.W., Berezin, S.: CVC Lite: A new implementation of the cooperating validity checker category B. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 515–518. Springer, Heidelberg (2004)
7. Barringer, H., Cheng, J.H., Jones, C.B.: A logic covering undefinedness in program proofs. Acta Informatica 21, 251–269 (1984)
8. Behm, P., Burdy, L., Meynadier, J.-M.: Well Defined B. In: Bert, D. (ed.) B 1998. LNCS, vol. 1393, pp. 29–45. Springer, Heidelberg (1998)
9. Berezin, S., Barrett, C., Shikanian, I., Chechik, M., Gurfinkel, A., Dill, D.L.: A practical approach to partial functions in CVC Lite. In: Workshop on Pragmatics of Decision Procedures in Automated Reasoning (2004)
10. Chalin, P.: Are the logical foundations of verifying compiler prototypes matching user expectations? Formal Aspects of Computing 19(2), 139–158 (2007)
11. Chalin, P.: A sound assertion semantics for the dependable systems evolution verifying compiler. In: ICSE, pp. 23–33. IEEE Computer Society Press, Los Alamitos (2007)
12. Cheng, J.H., Jones, C.B.: On the usability of logics which handle partial functions. In: Refinement Workshop, pp. 51–69 (1991)
13. Crow, J., Owre, S., Rushby, J., Shankar, N., Srivas, M.: A tutorial introduction to PVS. In: Workshop on Industrial-Strength Formal Specification Techniques (1995)
14. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
15. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs (2003)
16. Gries, D., Schneider, F.B.: Avoiding the undefined by underspecification. In: van Leeuwen, J. (ed.) Computer Science Today. LNCS, vol. 1000, pp. 366–373. Springer, Heidelberg (1995)
17. Guttag, J.V., Horning, J.J.: Larch: Languages and Tools for Formal Specification. Texts and Monographs in Computer Science. Springer, Heidelberg (1993)
18. Hähnle, R.: Many-valued logic, partiality, and abstraction in formal specification languages. Logic Journal of the IGPL 13(4), 415–433 (2005)
19. Hoogewijs, A.: On a formalization of the non-definedness notion. Zeitschrift für Mathematische Logik und Grundlagen der Mathematik 25, 213–217 (1979)
20. Jones, C.B.: Systematic software development using VDM. Prentice-Hall, Englewood Cliffs (1986)
21. Kiniry, J.R., Cok, D.R.: ESC/Java2: Uniting ESC/Java and JML. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 108–128. Springer, Heidelberg (2005)
22. Kleene, S.C.: On a notation for ordinal numbers. Journal of Symbolic Logic 3, 150–155 (1938)
23. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: a behavioral interface specification language for Java. ACM SIGSOFT Software Engineering Notes 31(3), 1–38 (2006)
24. McCarthy, J.: A basis for a mathematical theory of computation. In: Braffort, P., Hirschberg, D. (eds.) Computer Programming and Formal Systems, pp. 33–70. North-Holland, Amsterdam (1963)
25. Nipkow, T., Paulson, L.C., Wenzel, M.T.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)

26. Rudich, A., Darvas, Á., Müller, P.: Checking well-formedness of pure-method specifications. In: Cuellar, J., Maibaum, T. (eds.) Formal Methods (FM). LNCS, vol. 5014, pp. 68–83. Springer, Heidelberg (2008)
27. Rushby, J., Owre, S., Shankar, N.: Subtypes for Specifications: Predicate Subtyping in PVS. IEEE Transactions on Software Engineering 24(9), 709–720 (1998)
28. Schieder, B., Broy, M.: Adapting calculational logic to the undefined. The Computer Journal 42(2), 73–81 (1999)
29. Spivey, J.M.: Understanding Z: a specification language and its formal semantics. Cambridge University Press, Cambridge (1988)
30. Sutcliffe, G., Suttner, C.B., Yemenis, T.: The TPTP Problem Library. In: Bundy, A. (ed.) CADE 1994. LNCS, vol. 814, pp. 252–266. Springer, Heidelberg (1994)

A Semantic Proof of Equivalence

Structures. We define structures and interpretations in a way similar to as Behm et al. [8]. Let $\mathbf{A}$ be a set that does not contain $\perp$. We define $\mathbf{A}_\perp$ as $\mathbf{A} \cup \{\perp\}$. Let $\mathbf{F}$ be a set of function symbols, and $\mathbf{P}$ a set of predicate symbols. Let $\mathbf{I}$ be a mapping from $\mathbf{F}$ to the set of functions from $\mathbf{A}^n$ to $\mathbf{A}_\perp$, and from $\mathbf{P}$ to the set of predicates from $\mathbf{A}^n$ to $\{\mathbf{true}, \mathbf{false}\}$ (for simplicity, we assume that the interpretation of predicates is total), where n is the arity of the corresponding function or predicate symbol. We say that $\mathbf{M} = \langle \mathbf{A}, \mathbf{I} \rangle$ is a structure for our language with carrier set $\mathbf{A}$ and interpretation $\mathbf{I}$. We call a structure total if the interpretation of every function $\mathbf{f} \in \mathbf{F}$ is total, which means $\mathbf{f}(\ldots) \neq \perp$. We call the structure partial otherwise. A partial structure $\mathbf{M}$ can be extended to a total structure $\hat{\mathbf{M}}$ by having functions evaluated outside their domains return arbitrary values.

Interpretation. For a term $\mathbf{t}$, structure $\mathbf{M}$, and variable assignment $\mathbf{e}$, we denote the interpretation of $\mathbf{t}$ as $[\mathbf{t}]^{\mathbf{e}}_{\mathbf{M}}$. *Variable assignment* $\mathbf{e}$ maps the free variables of $\mathbf{t}$ to values. We define the interpretation of terms as given in Figure 6. Interpretation of formula φ denoted as $[\varphi]^{\mathbf{e}}_{\mathbf{M}}$ is given in Figure 7. **Dom** yields the domain of the interpretation of function symbols.
To check whether or not a value $\mathbf{l}$ is defined, we use function $\mathbf{wd}$:

$$\mathbf{wd}(\mathbf{l}) = \begin{cases} \mathbf{true}, & \text{if } \mathbf{l} \in \{\mathbf{true}, \mathbf{false}\} \\ \mathbf{false}, & \text{if } \mathbf{l} = \perp \end{cases}$$

$$[\,\mathbf{v}\,]^{\mathbf{e}}_{\mathbf{M}} \triangleq \mathbf{e}(\mathbf{v}) \text{ where } \mathbf{v} \text{ is a variable}$$

$$[\,\mathbf{f}(\mathbf{t}_1, \ldots, \mathbf{t}_n)\,]^{\mathbf{e}}_{\mathbf{M}} \triangleq \begin{cases} \mathbf{I}(\mathbf{f})([\mathbf{t}_1]^{\mathbf{e}}_{\mathbf{M}}, \ldots, [\mathbf{t}_n]^{\mathbf{e}}_{\mathbf{M}}), & \text{if } \langle [\mathbf{t}_1]^{\mathbf{e}}_{\mathbf{M}}, \ldots, [\mathbf{t}_n]^{\mathbf{e}}_{\mathbf{M}} \rangle \in \mathbf{Dom}(\mathbf{I}(\mathbf{f})) \\ & \quad \text{and } [\mathbf{t}_1]^{\mathbf{e}}_{\mathbf{M}} \neq \perp, \ldots, \ [\mathbf{t}_n]^{\mathbf{e}}_{\mathbf{M}} \neq \perp \\ \perp, \text{otherwise} \end{cases}$$

Fig. 6. Interpretation of terms

$$
\begin{array}{lcl}
[\,\mathit{true}\,]^{\mathbf{e}}_{\mathbf{M}} & \triangleq & \mathbf{true} \\
[\,\mathit{false}\,]^{\mathbf{e}}_{\mathbf{M}} & \triangleq & \mathbf{false} \\
[\,P(\mathbf{t}_1,\ldots,\mathbf{t}_n)\,]^{\mathbf{e}}_{\mathbf{M}} & \triangleq & \begin{cases} \mathbf{true}, & \text{if } \mathbf{I}(P)([\mathbf{t}_1]^{\mathbf{e}}_{\mathbf{M}},\ldots,[\mathbf{t}_n]^{\mathbf{e}}_{\mathbf{M}}) = \mathbf{true} \text{ and} \\ & \quad [\mathbf{t}_1]^{\mathbf{e}}_{\mathbf{M}} \neq \bot,\ldots,[\mathbf{t}_n]^{\mathbf{e}}_{\mathbf{M}} \neq \bot \\ \mathbf{false}, & \text{if } \mathbf{I}(P)([\mathbf{t}_1]^{\mathbf{e}}_{\mathbf{M}},\ldots,[\mathbf{t}_n]^{\mathbf{e}}_{\mathbf{M}}) = \mathbf{false} \text{ and} \\ & \quad [\mathbf{t}_1]^{\mathbf{e}}_{\mathbf{M}} \neq \bot,\ldots,[\mathbf{t}_n]^{\mathbf{e}}_{\mathbf{M}} \neq \bot \\ \bot, & \text{otherwise} \end{cases} \\
[\neg\varphi\,]^{\mathbf{e}}_{\mathbf{M}} & \triangleq & \begin{cases} \mathbf{true}, & \text{if } [\varphi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{false} \\ \mathbf{false}, & \text{if } [\varphi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{true} \\ \bot, & \text{otherwise} \end{cases} \\
[\,\varphi \wedge \phi\,]^{\mathbf{e}}_{\mathbf{M}} & \triangleq & \begin{cases} \mathbf{true}, & \text{if } [\varphi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{true} \text{ and } [\phi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{true} \\ \mathbf{false}, & \text{if } [\varphi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{false} \text{ or } [\phi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{false} \\ \bot, & \text{otherwise} \end{cases} \\
[\,\varphi \vee \phi\,]^{\mathbf{e}}_{\mathbf{M}} & \triangleq & \begin{cases} \mathbf{true}, & \text{if } [\varphi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{true} \text{ or } [\phi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{true} \\ \mathbf{false}, & \text{if } [\varphi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{false} \text{ and } [\phi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{false} \\ \bot, & \text{otherwise} \end{cases} \\
[\,\forall\, x.\ \varphi\,]^{\mathbf{e}}_{\mathbf{M}} & \triangleq & \begin{cases} \mathbf{true}, & \text{if for all } \mathbf{l} \in \mathbf{A},\ [\varphi]^{\mathbf{e}[x\leftarrow\mathbf{l}]}_{\mathbf{M}} = \mathbf{true} \\ \mathbf{false}, & \text{if there exists } \mathbf{l} \in \mathbf{A} \text{ such that} [\varphi]^{\mathbf{e}[x\leftarrow\mathbf{l}]}_{\mathbf{M}} = \mathbf{false} \\ \bot, & \text{otherwise} \end{cases} \\
[\,\exists\, x.\ \varphi\,]^{\mathbf{e}}_{\mathbf{M}} & \triangleq & \begin{cases} \mathbf{true}, & \text{if there exists } \mathbf{l} \in \mathbf{A} \text{ such that} [\varphi]^{\mathbf{e}[x\leftarrow\mathbf{l}]}_{\mathbf{M}} = \mathbf{true} \\ \mathbf{false}, & \text{if for all } \mathbf{l} \in \mathbf{A},\ [\varphi]^{\mathbf{e}[x\leftarrow\mathbf{l}]}_{\mathbf{M}} = \mathbf{false} \\ \bot, & \text{otherwise} \end{cases}
\end{array}
$$

Fig. 7. Interpretation of formulas

Lemma 1. *For every total structure* $\mathbf{M}$*, formula* φ*, and variable assignment* $\mathbf{e}$*, we have* $\mathbf{wd}([\varphi]^{\mathbf{e}}_{\mathbf{M}}) = \mathbf{true}$.

Proof. By induction over the structure of φ. □

Lemma 2. *For every structure* $\mathbf{M}$*, if* $\mathbf{M}$ *is extended to total structure* $\hat{\mathbf{M}}$*, then* $\mathbf{wd}([\varphi]^{\mathbf{e}}_{\hat{\mathbf{M}}}) = \mathbf{true}$.

Proof. Trivial consequence of the way $\mathbf{M}$ is extended and of **Lemma 1**. □

Domain Restrictions. Each function $\mathbf{f}$ is associated with a domain restriction d_f, which is a predicate that represents the domain of function $\mathbf{f}$. A structure $\mathbf{M}$ is a model for domain restrictions of functions in $\mathbf{F}$ (denoted by $d_{\mathbf{F}}(\mathbf{M})$) if and only if:

- The domain formulas are defined. That is, for each $\mathbf{f} \in \mathbf{F}$ and for all $\mathbf{e}$:
  $\mathbf{wd}([d_f]^{\mathbf{e}}_{\mathbf{M}}) = \mathbf{true}$
- Domain restrictions characterize the domains of function interpretations for $\mathbf{M}$. That is, for each $\mathbf{f} \in \mathbf{F}$ and $\mathbf{l}_1,\ldots,\mathbf{l}_n \in \mathbf{A}$:

$$[d_f]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{true} \quad \text{if and only if} \quad \langle \mathbf{l}_1, \dots, \mathbf{l}_n \rangle \in \mathbf{Dom}(\mathbf{I}(f))$$
where $e = [v_1 \to \mathbf{l}_1, \dots, v_n \to \mathbf{l}_n]$ and $\{v_1, \dots, v_k\}$ are $\mathbf{f}$'s parameter names.

In the following we prove two lemmas and finally our two main theorems.

Lemma 3. *For each structure* $\mathbf{M}$*, term* $\mathbf{t}$*, and variable assignment* $\mathbf{e}$*:*
if $d_{\mathbf{F}}(\mathbf{M})$ *then* $[\mathbf{t}]^{\mathbf{e}}_{\mathbf{M}} \neq \perp$ *if and only if* $[\delta(\mathbf{t})]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$*.*

Proof. By induction on the structure of $\mathbf{t}$ under the assumption that $d_{\mathbf{F}}(\mathbf{M})$.
Induction base: $\mathbf{t}$ is variable $\mathbf{v}$.
Since $[\mathbf{v}]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{e}(\mathbf{v}) \neq \perp$ and $[\delta(\mathbf{v})]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$, we have the desired property.
Induction step: $\mathbf{t}$ is function application $\mathbf{f}(\mathbf{t}_1, \dots, \mathbf{t}_n)$.
From definition of interpretation we get that $[\mathbf{f}(\mathbf{t}_1, \dots, \mathbf{t}_n)]^{\mathbf{e}}_{\mathbf{M}} \neq \perp$ if and only if:

$$\langle [\mathbf{t}_1]^{\mathbf{e}}_{\mathbf{M}}, \dots, [\mathbf{t}_n]^{\mathbf{e}}_{\mathbf{M}} \rangle \in \mathbf{Dom}(\mathbf{I}(\mathbf{f})) \ \wedge \ [\mathbf{t}_1]^{\mathbf{e}}_{\mathbf{M}} \neq \perp \wedge \dots \wedge [\mathbf{t}_n]^{\mathbf{e}}_{\mathbf{M}} \neq \perp$$

By the definition of $d_{\mathbf{F}}(\mathbf{M})$ and the induction hypothesis, it is equivalent to:

$$[d_f(\mathbf{t}_1, \dots, \mathbf{t}_n)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true} \wedge [\delta(\mathbf{t}_1)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true} \wedge \dots \wedge [\delta(\mathbf{t}_n)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$$

which is, by the definition of δ, equivalent to $[\delta(\mathbf{f}(\mathbf{t}_1, \dots, \mathbf{t}_n))]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$. □

Lemma 4. *For each structure* $\mathbf{M}$*, formula* φ*, and variable assignment* $\mathbf{e}$*:*
if $d_{\mathbf{F}}(\mathbf{M})$ *then* $[\varphi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{true}$ *if and only if* $[\mathcal{T}(\varphi)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$ *and*
$[\varphi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{false}$ *if and only if* $[\mathcal{F}(\varphi)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$*.*

Proof. By induction on the structure of φ under the assumption that $d_{\mathbf{F}}(\mathbf{M})$.
Induction base: φ is predicate $P(\mathbf{t}_1, \dots, \mathbf{t}_n)$.
From definition of interpretation we get $[P(\mathbf{t}_1, \dots, \mathbf{t}_n)]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{true}$ if and only if:

$$\mathbf{I}(P)([\mathbf{t}_1]^{\mathbf{e}}_{\mathbf{M}}, \dots, [\mathbf{t}_n]^{\mathbf{e}}_{\mathbf{M}}) = \mathbf{true} \ \wedge \ [\mathbf{t}_1]^{\mathbf{e}}_{\mathbf{M}} \neq \perp \wedge \dots \wedge [\mathbf{t}_n]^{\mathbf{e}}_{\mathbf{M}} \neq \perp$$

which is, by the assumption that the interpretation of predicates is total and by **Lemma 3**, equivalent to:

$$[P(\mathbf{t}_1, \dots, \mathbf{t}_n)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true} \wedge [\delta(\mathbf{t}_1)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true} \wedge \dots \wedge [\delta(\mathbf{t}_n)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$$

which is, by the definition of $\mathcal{T}$, equivalent to $[\mathcal{T}(P(\mathbf{t}_1, \dots, \mathbf{t}_n))]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$.
The proof is analogous for $\mathcal{F}$.
Induction step: For brevity, we only present the proof of those two cases for which the syntactic derivation was shown on page 105. The proofs are analogous for all other cases.

1. We prove that if $d_{\mathbf{F}}(\mathbf{M})$ then $[\gamma \wedge \phi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{true}$ if and only if $[\mathcal{T}(\gamma \wedge \phi)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$.
From definition of interpretation we get that $[\gamma \wedge \phi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{true}$ if and only if $[\gamma]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{true}$ and $[\phi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{true}$, which is, by the induction hypothesis, equivalent to $[\mathcal{T}(\gamma)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$ and $[\mathcal{T}(\phi)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$, which is, by the definition of $\mathcal{T}$, equivalent to $[\mathcal{T}(\gamma \wedge \phi)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$.

2. We prove that if $d_{\mathbf{F}}(\mathbf{M})$ then $[\forall\, x.\ \phi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{false}$ iff $[\mathcal{F}(\forall\, x.\ \phi)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$. From the definition of the interpretation function we get that $[\forall\, x.\ \phi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{false}$ if and only if there exists $\mathbf{l} \in \mathbf{A}$ such that $[\phi]^{\mathbf{e}[x \leftarrow \mathbf{l}]}_{\mathbf{M}} = \mathbf{false}$. By the induction hypothesis, this is equivalent to the existence of $\mathbf{l} \in \mathbf{A}$ such that $[\mathcal{F}(\phi)]^{\mathbf{e}[x \leftarrow \mathbf{l}]}_{\hat{\mathbf{M}}} = \mathbf{true}$, which is, by the definition of $\mathcal{F}$, equivalent to $[\mathcal{F}(\forall\, x.\ \phi)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$. □

Theorem 2. *For each structure* $\mathbf{M}$*, formula* φ*, and variable assignment* $\mathbf{e}$*:*
if $d_{\mathbf{F}}(\mathbf{M})$ *then* $\mathbf{wd}([\varphi]^{\mathbf{e}}_{\mathbf{M}}) = [\mathcal{Y}(\varphi)]^{\mathbf{e}}_{\hat{\mathbf{M}}}$

Proof. From the definition of **wd** we know that $\mathbf{wd}([\varphi]^{\mathbf{e}}_{\mathbf{M}})$ is defined. Furthermore, from **Lemma 2** (with φ substituted by $\mathcal{Y}(\varphi)$) we know that $[\mathcal{Y}(\varphi)]^{\mathbf{e}}_{\hat{\mathbf{M}}}$ is defined. Thus, it is enough to prove that $\mathbf{wd}([\varphi]^{\mathbf{e}}_{\mathbf{M}}) = \mathbf{true}$ if and only if $[\mathcal{Y}(\varphi)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$. Under the assumption that $d_{\mathbf{F}}(\mathbf{M})$, we have:

$\mathbf{wd}([\varphi]^{\mathbf{e}}_{\mathbf{M}}) = \mathbf{true}$ if and only if [by definition of **wd**]
$[\varphi]^{\mathbf{e}}_{\mathbf{M}} \in \{\mathbf{true}, \mathbf{false}\}$ if and only if [by **Lemma 4**]
$[\mathcal{T}(\varphi)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$ or $[\mathcal{F}(\varphi)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$ if and only if [by definition of $\mathcal{Y}$]
$[\mathcal{Y}(\varphi)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$ □

Berezin et al. [6] proved the following characteristic property of $\mathcal{D}$:

$$\text{if } d_{\mathbf{F}}(\mathbf{M}) \text{ then } \mathbf{wd}([\varphi]^{\mathbf{e}}_{\mathbf{M}}) = [\mathcal{D}(\varphi)]^{\mathbf{e}}_{\hat{\mathbf{M}}} \tag{2}$$

Theorem 3. *For each total structure* $\mathbf{M}$*, formula* φ*, and variable assignment* $\mathbf{e}$*:*
$[\mathcal{D}(\varphi) \Leftrightarrow \mathcal{Y}(\varphi)]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{true}$

Proof. For each total structure $\mathbf{M}$ there exists a partial structure $\mathbf{M}'$ such that $\mathbf{M} = \hat{\mathbf{M}}'$ and $d_{\mathbf{F}}(\mathbf{M}')$. We can build $\mathbf{M}'$ from $\mathbf{M}$ by restricting the domain of partial functions according to the domain restrictions.
By **Theorem 2** and (2) we get $[\mathcal{D}(\varphi)]^{\mathbf{e}}_{\hat{\mathbf{M}}'} = \mathbf{wd}([\varphi]^{\mathbf{e}}_{\mathbf{M}'}) = [\mathcal{Y}(\varphi)]^{\mathbf{e}}_{\hat{\mathbf{M}}'}$. Which is equivalent to $[\mathcal{D}(\varphi) \Leftrightarrow \mathcal{Y}(\varphi)]^{\mathbf{e}}_{\hat{\mathbf{M}}'} = \mathbf{true}$. Since $\mathbf{M} = \hat{\mathbf{M}}'$ we get the desired property. □

Proving Group Protocols Secure Against Eavesdroppers

Steve Kremer[1], Antoine Mercier[1], and Ralf Treinen[2]

[1] LSV, ENS Cachan, CNRS, INRIA, France
[2] PPS, Université Paris Diderot, CNRS, France

Abstract. Security protocols are small programs designed to ensure properties such as secrecy of messages or authentication of parties in a hostile environment. In this paper we investigate automated verification of a particular type of security protocols, called *group protocols*, in the presence of an eavesdropper, i.e., a passive attacker. The specificity of group protocols is that the number of participants is not bounded.

Our approach consists in representing an infinite set of messages exchanged during an unbounded number of sessions, one session for each possible number of participants, as well as the infinite set of associated secrets. We use so-called visibly tree automata with memory and structural constraints (introduced recently by Comon-Lundh et al.) to represent over-approximations of these two sets. We identify restrictions on the specification of protocols which allow us to reduce the attacker capabilities guaranteeing that the above mentioned class of automata is closed under the application of the remaining attacker rules. The class of protocols respecting these restrictions is large enough to cover several existing protocols, such as the GDH family, GKE, and others.

1 Introduction

Many modern computing environments, on wired or wireless networks, involve groups of users of variable size, and hence raise the need for secure communication protocols designed for an unbounded number of participants. This situation can be encountered in multi-user games, conferencing applications, or when securing an ad-hoc wireless network. In this paper we investigate the formal analysis of such protocols whose specification is parameterized by the number of participants. Proving protocols by hand is cumbersome and error-prone. Therefore we aim at automated proof methods. The variable number of protocol participants makes this task particularly difficult.

Related works. Several works already attempted to analyze these protocols. Steel developed the CORAL tool [15] which aims at searching for attacks. Pereira and Quisquater [14,13] analyzed specific types of protocols and proved the impossibility of building secure protocols using only some kinds of cryptographic primitives such as modular exponentiation in the presence of an active adversary. Küsters and Truderung [12,17] studied the automatic analysis of group protocols in case of a bounded number of sessions for an active intruder. They showed that the secrecy property is decidable for a given number of participants (without bound on this number) and for a group protocol that may be encoded in their model. As far as we know there is no complete and generic method to automatically prove the security of protocols for an unbounded number of participants. We aim to establish such a method.

Contribution of this paper. We consider a passive intruder, that is an intruder who only eavesdrops all the messages emitted during any sessions. This setting is of course restrictive in comparison to an active adversary. However, Katz and Yung [10] have shown that any group key agreement protocol which is secure against a passive adversary can be transformed into a protocol which resists against an active adversary. The security property that we are interested in here is the secrecy of some set of messages. In a group key exchange protocol, for example, the set of messages we hope remain secret is the set of session keys established during several sessions of the protocol.

Vulnerability of protocols designed for a *fixed* number of participants to eavesdropping attacks is quite straightforward to analyze in the symbolic model of Dolev and Yao. This is due to the fact that the set of messages exchanged during one session is a finite set, and that one can construct easily [9] a tree automaton describing precisely the set of messages an attacker can deduce from these using the deduction capabilities of the Dolev/Yao model. This is much more difficult in case of group protocols: not only does the number of messages exchanged during a protocol session grow with the number of participants, one also has to take into consideration parameters like the total number of participants of a session, and the position of a participant among the other protocol participants.

We specify a protocol as a function that assigns to any number of participants two sets of terms. The first set represents the set of messages emitted by the participants of the protocol, and the second set represents the set of terms that we want to remain secret. We suppose that the attacker has access to the set of messages exchanged by the legitimate protocol participants of the sessions for *all* possible numbers of participants combined, and that he attempts to deduce from this a supposed secret of *one* of the sessions. In other words, we are interested in the situation where an attacker could listen to several protocol sessions, say for 3, 4, and 5 participants, and then use this combined knowledge in order to deduce a supposed secret for one of these sessions.

As a first step we give sufficient restrictions on protocol specifications which allow us to reduce the intruder capabilities: we show that operations typically used in group protocols such as modular exponentiation and *exclusive or* are not necessary for the intruder to discover a secret term. As a consequence, the deduction capabilities of an intruder can be reduced to the classical so-called passive *Dolev-Yao* intruder. These restrictions are met by the protocols we have studied ([16,2,1], ...). In contrast to classical protocols between a fixed number of participants, however, we now have to deal with infinite sets of terms that are in general inductively defined.

The second step is to represent an over-approximation of the set of emitted messages and the set of supposed secrets in a formalism that has the following features:

- The set of messages that the intruder can deduce from the set of emitted messages can again be described by the same formalism.
- Disjointness of sets described in this formalism is decidable.

The formalism of classical tree automata enjoys these two properties; unfortunately it is not expressive enough to specify the inductively defined sets of messages that occur in group protocols. For this reason we employ here the recently [3] proposed class of so-called *visibly tree automata with memory, visibility and structural constraints*. Additionally, this constitutes a new unexpected application of this class of automata.

A further difficulty is that the messages may still use associative and commutative (AC) operators. The class of automata used here does not take into account AC operators. We will explain how to cope with this difficulty by working with certain representatives of equivalence classes modulo AC.

Structure of the paper. Section 2 presents the running example used throughout the paper. In Section 3 we introduce our attacker model. In Section 4 we explain the result on the reduction of intruder capabilities and its proof. In Section 5 we exhibit how to represent our sets of terms using the formalism of [3]. We illustrate our technique with the representation of the running example in this formalism in Section 6. We conclude in Section 7.

The details of the proofs are in the full version of this paper [11].

2 Running Example

Our running example is the group Diffie-Hellman key agreement protocol (GDH-2) [16]. Every participant in a session between n participants generates a nonce N_i (a secret fresh value). The first participant sends out $[\alpha, \alpha^{N_1}]$ where α is a publicly known constant. The i-th participant (for $1 < i < n$) expects from its predecessor a list of messages $[\alpha^{x_1}, \dots, \alpha^{x_i}]$, and he sends out $[\alpha^{x_i}, \alpha^{x_1 \cdot N_i}, \dots, \alpha^{x_i \cdot N_i}]$. The last participant, on reception of $[\alpha^{x_1}, \dots, \alpha^{x_n}]$, sends to all other participants $[\alpha^{x_1 \cdot N_n}, \dots, \alpha^{x_{n-1} \cdot N_n}]$. For instance in case of 4 participants the following sequence of messages is sent (we use here a list notation that will later be formalised by a binary pairing operator) :

Sender	Message
1	$[\alpha, \alpha^{N_1}]$
2	$[\alpha^{N_1}, \alpha^{N_2}, \alpha^{N_1 \cdot N_2}]$
3	$[\alpha^{N_1 \cdot N_2}, \alpha^{N_1 \cdot N_3}, \alpha^{N_2 \cdot N_3}, \alpha^{N_1 \cdot N_2 \cdot N_3}]$
4	$[\alpha^{N_1 \cdot N_2 \cdot N_4}, \alpha^{N_1 \cdot N_3 \cdot N_4}, \alpha^{N_2 \cdot N_3 \cdot N_4}]$

The common key is $\alpha^{N_1 \cdot N_2 \cdot N_3 \cdot N_4}$. Note that each of the participants i for $i < n$ can calculate that key from the message sent out by the last participant since he knows the missing part N_i, and that the participant n can calculate that key using the last element of the sequence he received form the participant $n - 1$.

$$\begin{aligned} x \oplus 0 &\to x \\ x \oplus x &\to 0 \\ ((x)^y)^z &\to x^{y \cdot z} \\ \langle x, y \rangle^z &\to \langle x^z, y^z \rangle \end{aligned}$$

Fig. 1. Rewrite System R

3 Model

We present here an extension of the model of a passive Dolev-Yao intruder [8]. This intruder is represented by a deduction system which defines his capabilities to obtain new terms from terms he already knows.

Messages. Messages exchanged during executions of the protocol are represented by terms over the following signature Σ:

$$\Sigma = \{\mathsf{pair}/2, \mathsf{enc}/2, \mathsf{exp}/2, \mathsf{mult}/2, \mathsf{xor}/2, H/1\} \uplus \Sigma_0$$

A pair $\mathsf{pair}(u, v)$ is usually written $\langle u, v \rangle$, the encryption of a message u by the key v, $\mathsf{enc}(u, v)$, is written $\{u\}_v$, and the exponentiation of u by v, $\mathsf{exp}(u, v)$, is written u^v. Multiplication mult and exclusive or xor are denoted by the infix operators $\cdot$ and $\oplus$. The symbol H denotes a unary hash function. Σ_0 is an infinite set of constant symbols, including nonces and encryption keys, and possibly elements of an algebraic structure such as the generator of some group. $\mathcal{T}(\Sigma)$ denotes the set of all terms build over Σ. We write $St(t)$ for the set of subterms of the term t, defined as usual, and extend this notation to sets of terms. We say that a function symbol f, where $f/n \in \Sigma$, *occurs in* a term t if $f(t_1, \ldots, t_n) \in St(t)$ for some $t_1, \ldots, t_n$.

Equational theory. We extend this model by an equational theory represented by the rewrite system R, which is given in Figure 1, modulo AC. The associative and commutative operators are *exclusive or* $\oplus$ and multiplication $\cdot$. Normalization by R modulo AC is denoted $\downarrow_{R/AC}$. The first two rules express the neutral element and the nilpotency law of *exclusive or*. The third rewrite rule allows for a normalization of nested exponentiations. Note that we do not consider a neutral element for multiplication, and that we do not have laws for multiplicative inverse, or for distribution of multiplication over other operations such as addition. The last rule allows one to normalize a list of terms exponentiated with the same term. It will be useful in order to model some protocols such as GKE [2]. Confluence and termination of this rewrite system have been proven using the tool C*i*ME [6]. In the rest of this paper we will only consider terms in normal form modulo R/AC. Any operation on terms, such as the intruder deduction system presented below, has to yield normal forms modulo R/AC.

The Intruder. The deduction capabilities of the intruder are described as the union of the deduction system DY of Figure 2 and the system of Figure 3. The

$$\frac{}{S \vdash t}\ \mathit{axiom} \text{ if } t \in S \qquad \frac{S \vdash t_1 \qquad S \vdash t_2}{S \vdash \langle t_1, t_2 \rangle}\ \mathit{pair}$$

$$\frac{S \vdash \langle t_1, t_2 \rangle}{S \vdash t_1}\ \mathit{proj}_1 \qquad \frac{S \vdash \langle t_1, t_2 \rangle}{S \vdash t_2}\ \mathit{proj}_2$$

$$\frac{S \vdash t_1 \qquad S \vdash t_2}{S \vdash \{t_1\}_{t_2}}\ \mathit{enc} \qquad \frac{S \vdash \{t_1\}_{t_2} \qquad S \vdash t_2}{S \vdash t_1}\ \mathit{dec}$$

$$\frac{S \vdash t}{S \vdash H(t)}\ \mathit{hash}$$

Fig. 2. The Dolev-Yao Deduction System DY

$$\frac{S \vdash t_1 \qquad S \vdash t_2 \text{ and } t_2 \in \Sigma_0}{S \vdash t_1^{t_2} \downarrow_{R/AC}}\ \mathit{exp} \qquad \frac{S \vdash t_1 \ \cdots \ S \vdash t_n}{S \vdash t_1 \oplus \cdots \oplus t_n \downarrow_{R/AC}}\ \mathit{Gxor}$$

Fig. 3. Extension of the Dolev-Yao Deduction System

complete system is called I. A sequent $S \vdash t$, where S is a finite set of terms and t a term, expresses the fact that the intruder can deduce t from S.

The system DY represents the classical Dolev-Yao intruder capacities to encrypt (*enc*) or decrypt (*dec*) a message with keys he knows, to build pairs (*pair*) of two messages, to extract one of the messages from a pair (*proj*$_1$, *proj*$_2$), and finally to apply the hash function (*hash*). Note that the term in the conclusion is in normal form w.r.t. R/AC when its hypotheses are, we hence do not have to normalize the term in the conclusion.

The system I extends the capabilities of DY by additional rules that allow the intruder to apply functions that are subject to the equational theory. We have to normalize the terms in the conclusions in these rules since applications of exponentiation or $\oplus$ may create new redexes.

As usually *pair*, *enc*, *hash*, *exp* and *Gxor* will be called *construction* rules, and *proj*$_1$, *proj*$_2$, and *dec*, will be called *deconstruction* rules. Note that the *Gxor* rule may also be used to deduce a subterm of a term thanks to the nilpotency of $\oplus$. For instance, let a and b be two constants of Σ_0. Applying a *Gxor* rule to sequents $S \vdash a \oplus b$ and $S \vdash b$ allows to deduce a, a subterm of $a \oplus b$. We implicitly assume that the constants 0 and α are always included in S.

For a set of ground terms S and a ground term t we write $S \vdash_D t$ if there exists a deduction of the term t from the set S in the deduction system D.

Definition 1 (Deduction). *A deduction of t from S in a system D is a tree where every node is labelled with a sequent. A node labelled with $S \vdash_D u$ has n sons $S \vdash_D v_1, \ldots, S \vdash_D v_n$ such that $\frac{S \vdash_D v_1, \ldots, S \vdash_D v_n}{S \vdash_D u}$ is an instance of one of the rules of D. The root is labelled by $S \vdash_D t$. The size of the deduction is the number of nodes of the tree.*

When the deduction system D is clear from the context we may sometimes omit the subscript D and just write $S \vdash t$. In the following we consider deductions in the systems DY and I. The *deductive closure* by a system D of a set of terms E is the set of all the terms the intruder can deduce from E using D.

Definition 2 (Deductive closure). *Let D be a deduction system and T a set of ground terms. The* deductive closure *by D of T is*

$$D(T) = \{t \mid T \vdash_D t\}$$

Protocol Specification and Secrecy Property. We suppose that a protocol is described by two functions $e : \mathbb{N} \to 2^{\mathcal{T}(\Sigma)}$ and $k : \mathbb{N} \to 2^{\mathcal{T}(\Sigma)}$. Given a number of participants n, $e(n)$ yields the set of terms that are emitted during a protocol execution with n participants and $k(n)$ is the set of secrets of this execution, typically the singleton set consisting of the constructed key.

Example 1. Consider our running example introduced Section 2. We have that

$$e_{\mathsf{GDH}}(n) = \begin{cases} \emptyset & \text{if } n < 2 \\ \{t_1^n, \ldots, t_n^n\} & \text{else} \end{cases} \quad \text{and} \quad k_{\mathsf{GDH}}(n) = \begin{cases} \emptyset & \text{if } n < 2 \\ \{\alpha^{N_1^n \cdot \ldots \cdot N_n^n}\} & \text{else} \end{cases}$$

where

$$\begin{aligned} t_1^n &= \langle \alpha, \alpha^{N_1^n} \rangle \\ t_i^n &= \langle \alpha^{N_1^n \cdot \ldots \cdot N_{i-1}^n}, t_{i-1}^{N_i^n} \rangle \qquad (1 < i < n) \\ t_n^n &= \langle \alpha^{N_2^n \cdot \ldots \cdot N_n^n}, \langle \ldots, \langle \alpha^{N_1^n \cdot \ldots \cdot N_{j-1}^n \cdot N_{j+1}^n \cdot \ldots \cdot N_n^n}, \langle \ldots, \alpha^{N_1^n \cdot \ldots \cdot N_{n-2}^n \cdot N_n^n} \rangle \rangle \rangle \rangle \end{aligned}$$

In the following we call a protocol specification the pair of (infinite) sets of terms (E, K) where $E = \bigcup_{n \in \mathbb{N}} e(n)$ and $K = \bigcup_{n \in \mathbb{N}} k(n)$. Given a protocol specification (E, K) we are interested in the question whether a supposed secret in K is deducible from the set of messages emitted in *any* of the sessions, i.e., $I(E) \cap K \overset{?}{=} \emptyset$. It is understood that $e(n)$ and $k(n)$, and hence E and K, are closed under associativity and commutativity of exclusive or and multiplication.

4 Reducing I to DY

In this section we show that, under carefully chosen restrictions on the sets E and K, we can consider an intruder who is weaker than expected. We will define *well-formed* protocols that will allow us to reduce a strong intruder (using the system I) to a weaker intruder (using only the system DY). This class is general enough to cover existing group protocols.

We first define a slightly stronger deduction system which will be more convenient for the proofs. Let R' be the rewrite system $R \setminus \{\langle x, y \rangle^z \to \langle x^z, y^z \rangle\}$, and I' the deduction system I where the rewrite system used in the rules *exp* and *Gxor* is R'. We show that any term which can be deduced in I can also be deduced in I'. This allows us to use the system I' which is more convenient for our proofs.

Lemma 1. *Let E be a set of terms. If $E \vdash_I t$ then $E \vdash_{I'} t$, and if $E \vdash_{I \setminus Gxor} t$ then $E \vdash_{I' \setminus Gxor} t$.*

To prove this result it is sufficient to note that each time an exponent is applied to a pair, one can obtain both elements of the pair by projection and apply the exponent to each of the elements before recomposing the pair.

We may however note that in general it is not the case that $E \vdash_{I'} t$ implies $E \vdash_I t$ as it is possible in I' to deduce a term of the form $\langle u, v\rangle^c$ (which would not be in normal form with respect to R).

Well formation. To state our well-formation condition, we define a closure function C on terms that computes an over-approximation of the constants that are deducible in a given term.

Definition 3 (closure). *Let $C : \mathcal{T}(\Sigma) \to 2^{\Sigma_0}$ be the function defined inductively as follows*

$$\begin{aligned} C(c) &= \{c\} \qquad \text{if } c \in \Sigma_0 \\ C(\langle u, v\rangle) &= C(u) \cup C(v) \\ C(\{u\}_v) &= C(u) \\ C(u_1 \oplus \cdots \oplus u_n) &= \textstyle\bigcup_{i=1} C(u_i) \\ C(f(u_1, \ldots, u_n)) &= \emptyset \qquad \text{if } f \neq \langle .,.\rangle, \{.\}. \end{aligned}$$

We extend this definition to sets of terms in the natural way, i.e., $c \in C(S)$ if there exists $u \in S$ such that $c \in C(u)$.

The following lemma states that if a constant c is in the closure of a term t which can be deduced from a set E, then c is also in the closure of E. A direct consequence is that if a constant c can be deduced from E then c is in the closure of E.

Lemma 2. *Let $c \in \Sigma_0$ and t be a term such that $c \in C(t)$. If $E \vdash_{I'} t$ then $c \in C(E)$.*

Corollary 1. *If $c \in \Sigma_0$ and $E \vdash_{I'} c$ then $c \in C(E)$.*

We impose restrictions on the protocols. These restrictions concern the usage of modular exponentiation and the usage of $\oplus$ during the execution of the protocol. We will also restrict the set of supposed secrets.

Definition 4 (Well formation). *A protocol specification (E, K) is said to be* well-formed *if it satisfies the following constraints.*

1. *If $t \in E$ and if $\oplus$ occurs in t, then $t = u \oplus v$ for some u and v and*
   – *$\oplus$ does not occur neither in u nor in v*
   – *$u, v \notin DY(E)$.*
2. *Let $t = u^{c_1 \cdot \cdots \cdot c_n}$ and $c_i \in \Sigma_0$ for all $1 \leq i \leq n$. If $t \in St(E \cup K)$ then $c_1, \ldots, c_n \notin C(E)$*
3. *For any $t \in K$, we have that $\oplus$ does not occur in t.*

Constraint 1 implies that $\oplus$ only occurs at the root position in terms of E and moreover $\oplus$ is of arity 2. Note that this constraint does not prevent the intruder from constructing terms where $\oplus$ has an arity greater than 2. Constraint 1 additionally requires that u and v cannot be deduced by a Dolev-Yao adversary. Constraint 2 requires that any constant occuring as an exponent in some term of $E \cup S$ cannot be accessed "easily", i.e. is not in the closure C, representing an over-approximation of accessible terms. Adding this constraint seems quite natural, since modular exponentiation is generally used to hide the exponent. This is for instance the case in the Diffie-Hellman protocol which serves as our running example: each participant of the protocol generates a nonce N, exponentiates some of the received values with N which are then send out. If the access to such a nonce would be "easy" then the computation of the established key would also be possible. Finally, constraint 3 requires that the secrets do not contain any xored terms.

These constraints allow us to derive the main theorem of this section and will be useful for automating the analysis of group protocols. While these constraints are obviously restrictive, they are nevertheless verified for several protocols, including GDH and GKE. In particular, the last constraint imposes that the specification of sets of keys must not involve any $\oplus$. This may seem rather arbitrary but the protocols we looked at fulfill this requirement.

The main theorem of this section states that, for well-formed protocols, if there is an attack for the attacker I then there is an attack for the attacker DY.

Theorem 1. *For all well-formed* (E, K) *we have* $I(E) \cap K = DY(E) \cap K$.

The proof of this result relies on several additional lemmas and is postponed to the end of this section. We only present some of the key lemmas. Remaining details and full proofs are given in [11] .

The following lemma is similar to Lemma 1 of [5], but adapted to our setting.

Lemma 3. *Let* E *be a set of terms. If* π *is a minimal deduction in* I' *of one of the following forms :*

$$\dfrac{\dfrac{\vdots}{E \vdash \langle u, v\rangle}}{E \vdash u}\,proj_1 \qquad \dfrac{\dfrac{\vdots}{E \vdash \langle u, v\rangle}}{E \vdash v}\,proj_2 \qquad \dfrac{\dfrac{\vdots}{E \vdash \{u\}_v} \quad \dfrac{\vdots}{E \vdash v}}{E \vdash u}\,dec$$

Then $\langle u, v\rangle \in St(E)$ *(resp.* $\langle u, v\rangle \in St(E)$*, resp.* $\{u\}_v \in St(E)$*).*

We now show that in the case of well-formed protocol specifications, if a deduction does not apply the *Gxor* rule, then it does not need to apply the *exp* rule neither.

Lemma 4. *Let* E *be a set of terms and* t *a term. If for every* $u^{c_1 \cdot \cdots \cdot c_n} \in St(E, t)$ *such that* $c_i \in \Sigma_0$ *one has* $c_i \notin C(E)$ *and if* $E \vdash_{I' \setminus Gxor} t$ *then* $E \vdash_{DY} t$.

The proof is done by induction on the length of the deduction tree showing that

$E \vdash_{I' \setminus Gxor} t$. The delicate case occurs when the last rule application is either projection or decryption. In these cases we rely on Lemma 3.

The next lemma states that whenever a *Gxor* rule is applied then a $\oplus$ occurs in the conclusion of the deduction. A direct corollary allows us to get rid of any application of the *Gxor* rule in well-formed protocol specifications.

Lemma 5. *Let E be a set of terms satisfying constraints 1 and 2 of Definition 4. Let π be a minimal deduction of $E \vdash_{I'} t$. If π involves an application of the Gxor rule, then $\oplus$ occurs in t.*

Corollary 2. *Let (E, K) be a well-formed protocol. Every minimal deduction of $E \vdash_{I'} t$ such that $t \in K$ does not involve an application of the Gxor rule.*

Proof. By contradiction. Let π be a minimal deduction of $E \vdash_{I'} t$ that involves a *Gxor* rule. As (E, K) is a well-formed protocol, by Lemma 5 $\oplus$ occurs in t. However, as $t \in K$ (E, K) is a well-formed protocol, by constraint 3, $\oplus$ does not occur in t.

We are now ready to prove the main theorem of this section.

Proof (of Theorem 1). We obviously have that $DY(E) \cap K \subseteq I(E) \cap K$ since DY is a subsystem of I. To prove the other direction, let t be a term of K and suppose that $E \vdash_I t$. By Lemma 1, there is a deduction of $E \vdash_{I'} t$. As (E, K) is well-formed and $t \in K$, by Corollary 2, there exists a deduction of $E \vdash_{I' \setminus Gxor} t$. Hence by Lemma 4, we have a deduction of $E \vdash_{DY} t$.

5 Representing Group Protocols by Automata

5.1 The Automaton Model

We first recall the definition of *Visibly Tree Automata with Memory and Structural Constraints* introduced first in [3] and later refined in [4]. Let $\mathcal{X}$ be an infinite set of variables. The set of terms built over Σ and $\mathcal{X}$ is denoted $T(\Sigma, \mathcal{X})$.

Definition 5 ([3]). *A* bottom-up tree automaton with memory *on a finite input signature Σ is a tuple (Γ, Q, Q_f, Δ) where Γ is a memory signature, Q is a finite set of unary state symbols, disjoint from $\Sigma \cup \Gamma$, $Q_f \subseteq Q$ is the subset of final states and Δ is a set of rewrite rules of the form $f(q_1(m_1), \ldots, q_n(m_n)) \to q(m)$ where $f \in \Sigma$ of arity n, $q_1, \ldots, q_n, q \in Q$ and $m_1, \ldots m_n, m \in T(\Sigma, \mathcal{X})$.*

For the next definition we will have to consider a partition of the signature, and we will also (as in [3]) require that all symbols of Σ and Γ have either arity 0 or 2. We also assume that Γ contains the constant $\perp$.

$$\Sigma = \Sigma_{\text{PUSH}} \uplus \Sigma_{\text{POP}_{11}} \uplus \Sigma_{\text{POP}_{12}} \uplus \Sigma_{\text{POP}_{21}} \uplus \Sigma_{\text{POP}_{22}}$$
$$\uplus \Sigma_{\text{INT}_0} \uplus \Sigma_{\text{INT}_1} \uplus \Sigma_{\text{INT}_2} \uplus \Sigma^{\equiv}_{\text{INT}_1} \uplus \Sigma^{\equiv}_{\text{INT}_2}$$

Two terms t_1 and t_2 are equivalent, written $t_1 \equiv t_2$, if they are equal when identifying all symbols of the same arity, that is $\equiv$ is the smallest equivalence on ground terms satisfying

- $a \equiv b$ for all a, b of arity 0,
- $f(s_1, s_2) \equiv g(t_1, t_2)$ if $s_1 \equiv t_1$ and $s_2 \equiv t_2$, for all f and g of arity 2.

Definition 6 ([4]). *A visibly tree automaton with memory and constraints (short* $\textsc{Vtam}^{\equiv}_{\not\equiv}$*) on a finite input signature* Σ *is a tuple* $(\Gamma, \equiv, Q, Q_f, \Delta)$ *where* Γ, Q, Q_f *are as in Definition 5,* $\equiv$ *is the relation on* $\mathcal{T}(\Gamma)$ *defined above and* Δ *is the set of rewrite rules of one of the following forms:*

$\textsc{Push}$	a	$\rightarrow$	$q(c)$	$a \in \Sigma_{\textsc{Push}}$
$\textsc{Push}$	$f(q_1(y_1), q_2(y_2))$	$\rightarrow$	$q(h(y_1, y_2))$	$f \in \Sigma_{\textsc{Push}}$
$\textsc{Pop}_{1i}$	$f(q_1(h(y_{11}, y_{12}), q_2(y_2)))$	$\rightarrow$	$q(y_{1i})$	$f \in \Sigma_{\textsc{Pop}_{1i}}, 1 \leq i \leq 2$
$\textsc{Pop}_{1i}$	$f(q_1(\perp), q_2(y_2))$	$\rightarrow$	$q(\perp)$	$f \in \Sigma_{\textsc{Pop}_{1i}}, 1 \leq i \leq 2$
$\textsc{Pop}_{2i}$	$f(q_1(y_1), q_2(h(y_{21}, y_{22})))$	$\rightarrow$	$q(y_{2i})$	$f \in \Sigma_{\textsc{Pop}_{2i}}, 1 \leq i \leq 2$
$\textsc{Pop}_{2i}$	$f(q_1(y_1), q_2(\perp))$	$\rightarrow$	$q(\perp)$	$f \in \Sigma_{\textsc{Pop}_{2i}}, 1 \leq i \leq 2$
$\textsc{Int}_0$	a	$\rightarrow$	$q(\perp)$	$a \in \Sigma_{\textsc{Int}_0}$
$\textsc{Int}_i$	$f(q_1(y_1), q_2(y_2))$	$\rightarrow$	$q(y_i)$	$f \in \Sigma_{\textsc{Int}_i}, 1 \leq i \leq 2$
$\textsc{Int}^{\equiv}_i$	$f(q_1(y_1), q_2(y_2))$	$\xrightarrow{y_1 \equiv y_2}$	$q(y_i)$	$f \in \Sigma_{\textsc{Int}^{\equiv}_i}, 1 \leq i \leq 2$
$\textsc{Int}^{\not\equiv}_i$	$f(q_1(y_1), q_2(y_2))$	$\xrightarrow{y_1 \not\equiv y_2}$	$q(y_i)$	$f \in \Sigma_{\textsc{Int}^{\equiv}_i}, 1 \leq i \leq 2$

where $q_1, q_2, q \in Q$*,* y_1*,* y_2 *are distinct variables of* $\mathcal{X}, c, h \in \Gamma$*.*

A $\textsc{Vtam}^{\equiv}_{\not\equiv}$ can apply a transition of type $\textsc{Int}^{\equiv}_i$ (resp. $\textsc{Int}^{\not\equiv}_i$) to a term $f(q_1(m_1), q_2(m_2))$ only when $m_1 \equiv m_2$ (resp. $m_1 \not\equiv m_2$). A term t is accepted by a $\textsc{Vtam}^{\equiv}_{\not\equiv}$ $\mathcal{A}$ in state $q \in Q$ and with memory $m \in T(\Gamma)$ iff $t \rightarrow^* q(m)$. The language $L(\mathcal{A}, q)$ and memory language $M(\mathcal{A}, q)$ of $\mathcal{A}$ in state q are respectively defined by:

$$L(\mathcal{A}, q) = \{t \mid \exists m \in T(\Gamma), t \rightarrow^* q(m)\} \qquad M(\mathcal{A}, q) = \{m \mid \exists t \in T(\Sigma), t \rightarrow^* q(m)\}$$

Theorem 2 ([3],[4]). *The class of languages recognizable by* $\textsc{Vtam}^{\equiv}_{\not\equiv}$ *is closed under Boolean operations, and emptiness of* $\textsc{Vtam}^{\equiv}_{\not\equiv}$ *is decidable.*

Note that the closure under $\cup$, $\cap$ supposes the same partition of the input signature Σ into $\Sigma_{\textsc{Push}}, \Sigma_{\textsc{Pop}_{11}}$ etc.

5.2 Encoding Infinite Signatures

The signature Σ used in the specification of the protocol may be infinite, in particular due to constants that are indexed by the number of a participants of a session. In order to be able to define $\textsc{Vtam}^{\equiv}_{\not\equiv}$ that recognize E and K we have to find an appropriate finite signature Σ' that contains only constants and binary symbols, and an appropriate function $\rho : T(\Sigma) \rightarrow T(\Sigma')$. The function ρ extends in a natural way to sets of terms. We will then use $\textsc{Vtam}^{\equiv}_{\not\equiv}$ constructions in order to show that $\rho(DY(E)) \cap \rho(K) = \emptyset$. Note that this implies $DY(E) \cap K = \emptyset$ independent of the choice of ρ, though in practice we will define ρ as an injective homomorphism. If ρ is injective then we have that disjointness of $DY(E)$ and K is equivalent to disjointness of $\rho(DY(E))$ and $\rho(K)$.

Example 2. The signature of our running example contains constants N_i^j, denoting the nonce of participant i in session j (where $i \leq j$). To make this example more interesting we could also consider constants K_i^j for symmetric keys between participants i and j (where $i < j$), and K_i^- (resp. K_i^+) for asymmetric decryption (resp. encryption) keys of the participant i.

We choose the finite signature Σ' consisting of the set of constants $\Sigma'_0 = \{0, \alpha\}$, and the set of binary function symbols

$$\Sigma'_2 = \{\mathsf{pair}, \mathsf{enc}, \mathsf{exp}, \mathsf{mult}, \mathsf{xor}, t, H, N, K, K^+, K^-, s, s'\}$$

The function $\rho\colon T(\Sigma) \to T(\Sigma')$ for the running example is defined as follows (using auxiliary functions $\rho_1\colon \mathbb{N} \to T(\Sigma')$ and $\rho_2\colon \{(i,j) \mid i \leq j\} \to T(\Sigma')$):

$$\begin{array}{rlrl}
\alpha & \to \alpha & \mathsf{pair}(u,v) & \to \mathsf{pair}(\rho(u), \rho(v)) \\
0 & \to 0 & \mathsf{enc}(u,v) & \to \mathsf{enc}(\rho(u), \rho(v)) \\
K_i^+ & \to K^+(0, \rho_1(i)) & \mathsf{exp}(u,v) & \to \mathsf{exp}(\rho(u), \rho(v)) \\
K_i^- & \to K^-(0, \rho_1(i)) & \mathsf{mult}(u,v) & \to \mathsf{mult}(\rho(u), t(0, \rho(v))) \\
K_i^j & \to K(0, \rho_2(i,j)) & \mathsf{xor}(u,v) & \to \mathsf{xor}(\rho(u), \rho(v)) \\
N_i^j & \to N(0, \rho_2(i,j)) & H(u) & \to H(0, \rho(u))
\end{array}$$

where we define

$$\begin{array}{ll}
\rho_1(i) = s'(0, \rho_1(i-1)) \text{ if } i > 0 & \rho_2(i,j) = s'(0, \rho_2(i-1, j-1)) \text{ if } i > 0 \\
\rho_1(0) = 0 & \rho_2(0,j) = s(0, \rho_2(0, j-1)) \quad \text{if } j > 0 \\
& \rho_2(0,0) = 0
\end{array}$$

For instance, $\rho_1(2) = s'(0, s'(0,0))$, and $\rho_2(1,3) = s'(0, s(0, s(0,0)))$. This encoding of pairs has been choosen in order to facilitate the automaton construction in Section 6.

Finally, we have to adapt the deduction system DY to the translation of the signature, yielding a modified deduction system DY' such that $\rho(DY(S)) = DY'(\rho(S))$ for any $S \subseteq T(\Sigma)$.

Example 3. (continued) In our running example we just have to adapt the rule *hash* and replace it by the following variant:

$$\frac{S \vdash t}{S \vdash H(0,t)}\ hash'$$

The other rules remain unchanged.

Lemma 6. *$\rho(DY(S)) = DY'(\rho(S))$ for ρ defined as in Example 2.*

The proof of this lemma can be found in [11] .

5.3 Coping with Associativity and Commutativity of xor and mult.

As for classical tree automata, the languages recognized by $\text{V}\textsc{tam}^{\equiv}_{\not\equiv}$ are in general not closed under associativity and commutativity. In order to cope with this difficulty we define a witness function W on $T(\Sigma')$ which associates to any term t the minimal element of the equivalence class $[t]_{AC}$ w.r.t. the order $\prec_{\Sigma'}$, the lexicographic path order [7] for the following precedence $<_{\Sigma'}$ on Σ':

$$0 <_{\Sigma'} \alpha <_{\Sigma'} s <_{\Sigma'} s' <_{\Sigma'} N <_{\Sigma'} K <_{\Sigma'} K^+ <_{\Sigma'}$$
$$K^- <_{\Sigma'} H <_{\Sigma'} t <_{\Sigma'} \mathsf{xor} <_{\Sigma'} \mathsf{mult} <_{\Sigma'} \mathsf{exp} <_{\Sigma'} \mathsf{enc} <_{\Sigma'} \mathsf{pair}$$

One verifies easily that $\rho(N_i^j) \prec_{\Sigma'} \rho(N_{i'}^{j'})$ if and only if either $i <_{\mathbb{N}} i'$, or $i = i'$ and $j <_{\mathbb{N}} j'$. We can now easily define the witness function:

Definition 7. *The function $W : T(\Sigma') \mapsto T(\Sigma')$ assigns to any $t' \in T(\Sigma')$ such that $t' = \rho(t)$ the minimal element of $\rho([t]_{AC})$.*

This function extends in a natural way to sets of terms. Now, the disjointness of two sets of terms S_1 and S_2 that are closed under congruence modulo AC is equivalent to the disjointness of $W(S_1)$ and $W(S_2)$.

Theorem 3. *If S is closed under AC then $W(DY'(S)) = DY'(W(S))$.*

5.4 Closure under DY and Compatibility with the Closure under AC

Theorem 4. *For every* $\mathrm{VTAM}_{\not\equiv}^{\equiv}$ $\mathcal{A}$, *such that* $\mathsf{pair}, \mathsf{enc} \notin \{\Sigma'_{\mathrm{INT}_1^{\equiv}} \cup \Sigma'_{\mathrm{INT}_2^{\equiv}}\}$ *and the only constant symbol of Γ is $\bot$, there exists a* $\mathrm{VTAM}_{\not\equiv}^{\equiv}$ $\mathcal{A}_{DY}$ *such that $L(\mathcal{A}_{DY}) = DY'(L(\mathcal{A}))$.*

The proof is based on the classical technique of completion of the automaton (see [9]), with special care taken to the extension to memory and constraints. The complete construction is given in [11] and depends on the partition of the input signature, we illustrate it here for the case where $\mathsf{pair}, \mathsf{enc}, H \in \Sigma_{\mathrm{PUSH}}$. The automaton extends $\mathcal{A}$ by new final states q_{pair}, q_{enc}, and q_H. We also add some new transitions and promote some states to final states:

$$\frac{q_1, q_2 \in Q_f}{\mathsf{pair}(q_1(x), q_2(y)) \to q_{pair}(h(x,y))}\ Pair$$

$$\frac{\mathsf{pair}(q_1(x), q_2(y)) \to q(h(x,y)) \qquad q \in Q_f \qquad L(\mathcal{A}, q_{3-i}) \neq \emptyset}{q_i \in Q_f}\ Proj_i,\ 1 \leq i \leq 2$$

$$\frac{q_1, q_2 \in Q_f}{\mathsf{enc}(q_1(x), q_2(y)) \to q_{enc}(h(x,y))}\ Enc$$

$$\frac{\mathsf{enc}(q_1(x), q_2(y)) \to q(h(x,y)) \qquad q \in Q_f \qquad L(\mathcal{A}, q_2) \cap L(\mathcal{A}) \neq \emptyset}{q_1 \in Q_f}\ Dec$$

$$\frac{q_1 \in Q_f}{H(q_0(x), q_1(y)) \to q_H(h(x,y))}\ Hash$$

6 Example

Here we propose an over-approximation of the set of computed keys during an unbounded number of sessions of the protocol (one session for each number of participants). An over-approximation of the set of emitted messages and its representation by automata is given in [11] .

The approximation we propose to represent is the following:

$$K = \{\alpha^{N_{j_1}^{j_1} \cdot N_{(j_1-1)}^{j_2} \cdots N_1^{j_{j_1}}}\}$$

Here we only give the construction of the automaton $\mathcal{A}_K$ recognizing the set K. K is the set of symbols of the form α^p where p is a product of nonces N_i^j :

– $i = j$ for the maximal nonce N_i^j in p,
– the number of nonces is j, where N_i^j is the maximal nonce,
– for every i such that $1 \le i \le j$, N_i^k belongs to p for some k.

We use the following partition of the signature Σ' in the automata:

$$\Sigma_{PUSH} = \{s', \mathsf{exp}, 0, \alpha\} \qquad \Sigma_{POP_{22}} = \{t\}$$
$$\Sigma_{INT_2^=} = \{\mathsf{mult}\} \qquad \Sigma_{INT_2} = \{s, N\}$$

The other symbols can be put into any part of the signature. We define $\Gamma = \{S, S', h, \bot\}$. The automaton $\mathcal{A}_K$ is defined as follows (q_{acc} is the final state):

$$0 \to q_d(\bot) \qquad \alpha \to q_\alpha(\bot)$$

The following transitions check that if a term $t \to^* q_{nent}(m)$ then t is of the form $N(0, s'(0, \ldots s'(0, 0) \ldots))$ and $m = S'(0, \ldots S'(0, 0) \ldots)$ and the number of S' equals the number of s'. Hence t represents a nonce such that $i = j$.

$$\begin{aligned} s'(q_d(m), q_d(m')) &\to q_{s'ent}(S'(m, m')) \\ s'(q_d(m), q_{s'ent}(m')) &\to q_{s'ent}(S'(m, m')) \\ N(q_d(m), q_{s'ent}(m')) &\to q_{nent}(m') \end{aligned}$$

The following transitions are similar but also allow several S between the S' and the constant 0. We count in the memory only the number of S'. We also check that terms leading to $q_{nonly1s'}$ involve at most one occurrence of the symbol s'.

$$\begin{aligned} s(q_d(m), q_d(m')) &\to q_d(m') \\ s'(q_d(m), q_d(m')) &\to q_{only1s'}(S'(m, m')) \\ s'(q_d(m), q_{only1s'}(m')) &\to q_{s'}(S'(m, m')) \\ s'(q_d(m), q_{s'}(m')) &\to q_{s'}(S'(m, m')) \\ N(q_d(m), q_{s'}(m')) &\to q_n(m') \\ N(q_d(m), q_{only1s'}(m')) &\to q_{nonly1s'}(m') \end{aligned}$$

The following transitions remove an S' symbol from the memory.

$$\begin{aligned} t(q_d(m), q_{nent}(S'(m', m''))) &\to q_{nt}(m'') \\ t(q_d(m), q_{narg}(S'(m', m''))) &\to q_{nt}(m'') \end{aligned}$$

The following transition can be applied between a term that represents a nonce and a term that represents either a product or a nonce n_{jj} on which we have applied one af the above transitions.

$$\mathsf{mult}(q_n(m), q_{nt}(m')) \overset{m \equiv m'}{\rightarrow} q_{narg}(m)$$

The following transition applies (by the memory language of $q_{nonly1s'}$) only if m' is $S'(\perp, \perp)$. In this case the term is considered a possible product of K.

$$\mathsf{mult}(q_{nonly1s'}(m), q_{nt}(m')) \overset{m \equiv m'}{\rightarrow} q_{exp}(m)$$

In this case it is possible to apply this last transition.

$$\mathsf{exp}(q_\alpha(m), q_{exp}(m')) \rightarrow q_{acc}(h(m, m'))$$

The following lemma states that our automaton recognizes in fact a slight over-appoximation of $W(\rho(K))$ as it recognizes also some terms that are not witnesses (but that are still in $\rho(K)$).

Lemma 7. *$W(\rho(K)) \subseteq L(\mathcal{A}_K) \subseteq \rho(K)$.*

Lemma 8. *$(L(\mathcal{A}_{E_1}) \cup L(\mathcal{A}_{E_2}), L(\mathcal{A}_K))$ is well-formed.*

Proof. As no transitions has a left hand side headed by an xor, constraints (1) and (3) of Definition 4 are satisfied. We can check on the construction of the automata $\mathcal{A}_{E_1}$ and $\mathcal{A}_{E_2}$ (given in [11]) that every term t accepted by these automata is of the form $\mathsf{exp}(u, v)$ for some u and v. By Definition 3, this implies that $C(L(\mathcal{A}_{E_1}) \cup L(\mathcal{A}_{E_2})) = \emptyset$.

7 Conclusion

We have shown that for a class of well-formed protocols, a general model of intruder capabilities including applications of modular exponentiation and exclusive or is equivalent to a weaker model which can be seen as the classical Dolev-Yao model modulo associativity and commutativity of some operators. We have then shown, by a series of reductions and over-approximations, that the secrecy problem for group protocols in presence of a passive attacker can be shown by using advanced tree automata techniques. We have shown how to check (over-approximations of) conditions on the indexes of constants appearing in a term by a $\mathrm{V{\small TAM}}^{\equiv}_{\not\equiv}$ automaton, how to cope with congruence classes modulo associativity and commutativity in this automata model, and finally that recognizability by this class of automata is preserved by construction of the Dolev-Yao closure.

While our approach applies to several examples of group protocols there is still room for improvements. The first possible generalization concerns our definition of well-formation of a group protocol. Some of the clauses of our definition seem to be rather natural, whereas some others are more arbitrary. A possible continuation of this work is to relax or to modify some of these restrictions, keeping in mind that it must still be possible to prove a reduction result to the classical Dolev-Yao intruder model.

Another restriction of our approach consists in the hypothesis that the only possible exponents are products of constants. This is not the case in general. Group protocols involving exponents different from a simple product exist. An exponent could be represented by a sum, or by an exponentiation itself. The theory of modular exponentiation seems to remain hard to manage in its full generality.

An important avenue of future research is the automatisation of the construction of the automaton recognizing the set of emitted messages, resp. of supposed secrets. This includes the definition of a specification language proper to group protocols.

References

1. Boyd, C., González Nieto, J.-M.: Round-optimal ciontributory conference key agreement. In: Desmedt, Y.G. (ed.) PKC 2003. LNCS, vol. 2567, pp. 161–174. Springer, Heidelberg (2002)
2. Bresson, E., Chevassut, O., Essiari, A., Pointcheval, D.: Mutual authentication and group key agreement for low-power mobile devices. Computer Communications 27(17), 1730–1737 (2004)
3. Comon-Lundh, H., Jacquemard, F., Perrin, N.: Tree automata with memory, visibility and structural constraints. In: Seidl, H. (ed.) FOSSACS 2007. LNCS, vol. 4423, pp. 168–182. Springer, Heidelberg (2007)
4. Comon-Lundh, H., Jacquemard, F., Perrin, N.: Visibly tree automata with memory and constraints. Research Report LSV-07-30, Laboratoire Spécification et Vérification, ENS Cachan, France, Logical Methods in Computer Science (September 2007) (to appear)
5. Comon-Lundh, H., Shmatikov, V.: Intruder deductions, constraint solving and insecurity decision in presence of exclusive or. In: Proceedings of the 18th IEEE Symposium on Logic in Computer Science (LICS 2003), vol. 171, pp. 271–280. IEEE Computer Society Press, Los Alamitos (2003)
6. Contejean, E., Marché, C., Monate, B., Urbain, X.: The CiME Rewrite Tool (2000), http://cime.lri.fr
7. Dershowitz, N.: Termination of rewriting. J. Symb. Comput. 3(1-2), 69–116 (1987)
8. Dolev, D., Yao, A.C.: On the security of public key protocols. IEEE Transactions on Information Theory 29(2), 198–208 (1983)
9. Goubault-Larrecq, J.: A method for automatic cryptographic protocol verification (extended abstract). In: IPDPS-WS 2000. LNCS, vol. 1800, pp. 977–984. Springer, Heidelberg (2000)
10. 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)
11. Kremer, S., Mercier, A., Treinen, R.: Proving group protocols secure against eavesdroppers. Research Report LSV, Laboratoire Spécification et Vérification, ENS Cachan, France (May 2008), http://www.lsv.ens-cachan.fr/Publis/RAPPORTS_LSV/rapports.php?filename=lsv-2008

12. Küsters, R., Truderung, T.: On the Automatic Analysis of Recursive Security Protocols with XOR. In: Thomas, W., Weil, P. (eds.) STACS 2007. LNCS, vol. 4393. Springer, Heidelberg (2007)
13. Pereira, O., Quisquater, J.-J.: Some attacks upon authenticated group key agreement protocols. Journal of Computer Security 11(4), 555–580 (2003)
14. Pereira, O., Quisquater, J.-J.: On the impossibility of building secure cliques-type authenticated group key agreement protocols. Journal of Computer Security 14(2), 197–246 (2006)
15. Steel, G., Bundy, A.: Attacking group protocols by refuting incorrect inductive conjectures. Journal of Automated Reasoning 36(1-2), 149–176 (2006)
16. Steiner, M., Tsudik, G., Waidner, M.: Diffie-Hellman key distribution extended to group communication. In: ACM Conference on Computer and Communications Security, pp. 31–37 (1996)
17. Truderung, T.: Selecting theories and recursive protocols. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 217–232. Springer, Heidelberg (2005)

Automated Implicit Computational Complexity Analysis (System Description)*

Martin Avanzini[1], Georg Moser[2], and Andreas Schnabl[2]

[1] Master Program in Computer Science, University of Innsbruck, Austria
martin.avanzini@student.uibk.ac.at
[2] Institute of Computer Science, University of Innsbruck, Austria
{georg.moser,andreas.schnabl}@uibk.ac.at

Abstract. Recent studies have provided many characterisations of the class of polynomial time computable functions through term rewriting techniques. In this paper we describe a (fully automatic and command-line based) system that implements the majority of these techniques and present experimental findings to simplify comparisons.

1 Introduction

Recent studies have provided many characterisations of the class of polynomial time computable functions. Our main interest lies in studies that employ term rewriting as abstract model of computation and consequently use existing techniques from rewriting to characterise several computational complexity classes, cf. [1,2,3,4,5,6]. The use of rewriting techniques opens the way for automatisation. In this paper, we present a fully automatic complexity tool that implements a majority of the techniques introduced in the cited literature. We describe the system and its implementation in some detail, and report on experimental results comparing the relative strength of the implemented techniques.

For brevity and greater applicability we consider techniques that directly classify runtime complexity, i.e., those techniques use the number of rewrite steps (perhaps allowing for a specific rewrite strategy) as complexity measure, namely *additive polynomial interpretations* [2] and *polynomial path orders* [6]. In Section 2, we recall the main theorems for these proof methods, and describe their implementation in some detail. Apart from that, we have also implemented the *light multiset path order* [3] (*LMPO* for short) and *quasi-interpretations* in conjunction with the *product path order* [1,7,8] (RPO^{QI}_{Pro} for short) in our tool. Opposed to the former methods, the latter two rely on non-trivial evaluation techniques, namely caching of intermediate results.

In order to compare the different methods proposed in the literature, we tested their applicability on (subsets of) the Termination Problem Data Base (TPDB for short), which is used in the annual termination competition. Arguably this is an imperfect choice as the TPDB has been designed to test the strength of termination provers in rewriting, not as a testbed to analyse the implicit computational complexity of TRSs. On the other hand, it is the only (relatively

* This research is supported by FWF (Austrian Science Fund) project P20133.

large) collection of TRSs that is publicly available and we have taken some effort in selecting interesting and meaningful subsets of this collection.

It seems natural that *additive polynomial interpretations* and *polynomial path orders*, which only use rewriting as an underlying model of computation, perform worse in comparison to *quasi-interpretations* and the *light multiset path order*. Interestingly, our practical findings cannot confirm this: on our studied testbeds, the former techniques outperform the latter two if we count the total number of TRSs that can be handled. Furthermore, a closer look on the testbeds reveals that every single one of the implemented techniques can handle at least one example that cannot be handled by any other method (see Section 4 for more details). This implies that a notable part of our testbeds is covered with the union of the implemented methods.

2 Methods That Directly Classify Polytime

In this section, we describe *additive polynomial interpretations* and *polynomial path orders* in more detail. To keep this system description short, we assume familiarity with rewriting (see for example [9]). Our first task is to fix what is meant by the function *computed* by a TRS. We write $s \rightarrow^{!}_{\mathcal{R}} t$ for $s \rightarrow^{*}_{\mathcal{R}} t$, whenever t is a normal form with respect to a TRS $\mathcal{R}$. Suppose $\ulcorner \cdot \urcorner$ is an encoding function and let $\mathcal{R}$ denote a completely defined and orthogonal constructor TRS. Let Σ be an alphabet. An n-ary function $f\colon (\Sigma^*)^n \rightarrow \Sigma^*$ is *computable* by $\mathcal{R}$ if there exists a defined function symbol f such that for all $w_1, \dots, w_n, v \in \Sigma^*$: $\mathsf{f}(\ulcorner w_1 \urcorner, \dots, \ulcorner w_n \urcorner) \rightarrow^{!} \ulcorner v \urcorner$ if and only if $f(w_1, \dots, w_n) = v$. On the other hand we say that $\mathcal{R}$ *computes* f if the function $f\colon (\Sigma^*)^n \rightarrow \Sigma^*$ is defined by the above equation. We say that a TRS $\mathcal{R}$ has a *simple signature* (cf. [3]) if we can type all constructors of $\mathcal{R}$ and define a mapping S from the set of sorts to $\mathbb{N}$ such that for every constructor of sort $s_1, \dots, s_n \rightarrow s$, there exists an i with $S(s_i) \leqslant S(s)$, and $S(s_j) < S(s)$ for all $j \neq i$. The *(innermost) runtime complexity function* with respect to a TRS $\mathcal{R}$ relates the length of the longest (innermost) derivation sequence in $\mathcal{R}$ to the size of the arguments of the initial term of the form $f(t_1, \dots, t_n)$, where f is a defined function symbol, and t_i are ground terms consisting only of constructor symbols.

A suitable starting point for the classification of polytime computable functions is to look at *polynomial interpretations*. However, as shown in [10], polynomial interpretations without further restrictions still admit a double exponential upper bound on the runtime complexity of a given TRS. Hence restricted polynomial interpretations are studied in the literature. A polynomial $P(x_1, \dots, x_n)$ (over $\mathbb{N}$) is called *additive* (cf. [2]) if $P(x_1, \dots, x_n) = x_1 + \cdots + x_n + a$ where $a \in \mathbb{N}$. A polynomial interpretation $\mathcal{A}$ is called *simple-mixed, constructor-restricted* (*SMC* for short) if all interpretation functions $f_{\mathcal{A}}$ are additive polynomials whenever f is a constructor symbol, and *simple-mixed polynomials* (cf. [11]) otherwise. The following theorem is shown in [2, Lemma 3, Theorem 4].

Theorem 1. *Let $\mathcal{R}$ be a finite constructor TRS that is compatible with an SMC-interpretation. Then the runtime complexity with respect to $\mathcal{R}$ is polynomial.*

Furthermore, the functions computable by a finite and orthogonal constructor TRS that is compatible with an SMC-interpretation are polytime computable.

Recently in [6] a restriction of the multiset path order, called *polynomial path order* (*POP** for short) has been introduced. In [6] POP* is defined for a strict precedence. Below we extend this definition to quasi-precedences $\gtrsim$, whose equivalence part is denoted as $\sim$. We require that the arity of f equals the arity of g whenever $f \sim g$. POP* relies on the separation of *safe* and *normal* inputs. A safe mapping safe associates with every n-ary function symbol f the set of *safe argument positions.* If f is a defined function symbol then $\mathsf{safe}(f) \subseteq \{1, \ldots, n\}$, otherwise we fix $\mathsf{safe}(f) = \{1, \ldots, n\}$. The argument positions not included in $\mathsf{safe}(f)$ are called *normal* and denoted by $\mathsf{nrm}(f)$. Let $\gtrsim$ be a quasi-precedence and safe a safe mapping. The polynomial path order $\gtrsim_{\mathsf{pop}*}$ is an extension of the auxiliary order $\gtrsim_{\mathsf{pop}}$. In order to define $\gtrsim_{\mathsf{pop}*}$ we give the definition of its strict part $\succ_{\mathsf{pop}*}$ and its equivalence part $\sim_{\mathsf{pop}*}$ separately. We define $f(s_1, \ldots, s_n) \sim_{\mathsf{pop}*} g(t_1, \ldots, t_n)$ (respectively $f(s_1, \ldots, s_n) \sim_{\mathsf{pop}} g(t_1, \ldots, t_n)$) if and only if $f \sim g$, and the safe and normal part of the multisets $[s_1, \ldots, s_n]$ and $[t_1, \ldots, t_n]$ are equivalent over $\sim_{\mathsf{pop}*}$ (respectively $\sim_{\mathsf{pop}}$). We define the order $\succ_{\mathsf{pop}}$ inductively as follows: $s = f(s_1, \ldots, s_n) \succ_{\mathsf{pop}} t$ if either

1. $f \in \mathcal{C}$ and $s_i \gtrsim_{\mathsf{pop}} t$ for some $i \in \{1, \ldots, n\}$, or
2. $s_i \gtrsim_{\mathsf{pop}} t$ for some $i \in \mathsf{nrm}(f)$, or
3. $t = g(t_1, \ldots, t_m)$ with $f \in \mathcal{D}$, $f \succ g$, and $s \succ_{\mathsf{pop}} t_i$ for all $1 \leqslant i \leqslant m$.

Here $\mathcal{D}$ ($\mathcal{C}$) denotes the defined (constructor) symbols, respectively. Based on $\succ_{\mathsf{pop}}$ we define $\succ_{\mathsf{pop}*}$ inductively as follows: $s = f(s_1, \ldots, s_n) \succ_{\mathsf{pop}*} t$ if either

1. $s \succ_{\mathsf{pop}} t$, or
2. $s_i \gtrsim_{\mathsf{pop}*} t$ for some $i \in \{1, \ldots, n\}$, or
3. $t = g(t_1, \ldots, t_m)$, with $f \in \mathcal{D}$, $f \succ g$, and the following properties hold: (i) $s \succ_{\mathsf{pop}*} t_{i_0}$ for some $i_0 \in \mathsf{safe}(g)$ and (ii) either $s \succ_{\mathsf{pop}} t_i$ or $s \rhd t_i$ and $i \in \mathsf{safe}(g)$ for all $i \neq i_0$, or
4. $t = g(t_1, \ldots, t_n)$, $f \sim g$ and for $\mathsf{nrm}(f) = \{i_1, \ldots, i_p\}$, $\mathsf{safe}(f) = \{j_1, \ldots, j_q\}$), $\mathsf{nrm}(g) = \{i'_1, \ldots, i'_{p'}\}$, $\mathsf{safe}(g) = \{j'_1, \ldots, j'_{q'}\}$, both $[s_{i_1}, \ldots, s_{i_p}]$ $(\succ_{\mathsf{pop}*})_{\mathsf{mul}}$ $[t_{i'_1}, \ldots, t_{i'_{p'}}]$ and $[s_{j_1}, \ldots, s_{j_q}]$ $(\gtrsim_{\mathsf{pop}*})_{\mathsf{mul}}$ $[t_{j'_1}, \ldots, t_{j'_{q'}}]$ hold.

Here $(\succ)_{\mathsf{mul}}$ denotes the multiset extension of an order $\succ$. We arrive at the following theorem, which is an easy consequence of the main results from [6].

Theorem 2. *Let $\mathcal{R}$ be a finite constructor TRS compatible with $\succ_{\mathsf{pop}*}$, i.e., $\mathcal{R} \subseteq \succ_{\mathsf{pop}*}$. Then the induced innermost runtime complexity is polynomial. Furthermore, the functions computable by a finite and orthogonal constructor TRS with a simple signature that is compatible with an instance of POP* are exactly the polytime computable functions.*

In order to increase the applicability of POP*, we employ semantic labeling [12] with Boolean models. It is straightforward to check that the maximal lengths of derivations in the original and the labeled system are equal. In order to apply Theorem 2 it is mandatory to restrict to *finite* models. We discuss the implementation of POP* with semantic labeling below in Section 3.

3 Implementation

The here presented fully automatic system `icct`[1] implements the techniques introduced in [1,2,3,6,7]. In addition to the techniques described in Section 2, our system `icct` can handle the methods LMPO [3] and RPO^{QI}_{Pro} [1]. We have not (yet) considered (quasi-friendly) sup-interpretations, cf. [4,5], but plan to do so in the future. Furthermore our system `icct` only searches for additive polynomial quasi-interpretations, without the max-function. This contrasts with the system `crocus`[2] implementing quasi-interpretations as defined in [8]. We also plan to add this in the future, possibly using the methods described in [13].

Our implementation is (partly) based on the termination prover $\mathsf{T_TT_2}$[3] and therefore our (command-line) interface is compatible with $\mathsf{T_TT_2}$. The modular design of our implementation allows for an immediate integration of the presented system as a plug-in to $\mathsf{T_TT_2}$. Like $\mathsf{T_TT_2}$, our methods are written fully in `OCaml`, and the system `icct` extends $\mathsf{T_TT_2}$ by around 3500 lines of code. In the remainder of this section, we describe the implementation of SMC and POP*.

In order to search for SMC interpretations, we follow well-established methods stemming from termination analysis. Our starting point is the use of abstract polynomial interpretations, cf. [14]. For SMC, that is: $f_{\mathcal{A}}(x_1,\ldots,x_n) = \sum_{i_j \in \{0,1\}} a_{i_1,\ldots,i_n} x_1^{i_1} \cdot \ldots \cdot x_n^{i_n} + \sum_{i=1}^{n} b_i x_i^2$ if $f \in \mathcal{D}$, and $f_{\mathcal{A}}(x_1,\ldots,x_n) = \sum_{i=1}^{n} x_i + c$ otherwise. The variables $a_{i_1,\ldots,i_n}$, b_i, and c are called *coefficient variables*. Given such abstract interpretations we transform the compatibility and monotonicity tests into Diophantine (in)equalities in the coefficient variables. Putting an upper bound on the coefficient variables makes the problem finite, allowing it to be transformed into SAT. The encoding into SAT essentially follows [15]. To actually solve the satisfiability problem, MiniSAT[4] is invoked. A satisfying assignment is used to instantiate the coefficient variables suitably.

To prove compatibility of a given TRS $\mathcal{R}$ with polynomial path orders we have to find a quasi-precedence $\succsim$ and a *safe mapping* such that the induced order is compatible with $\mathcal{R}$. Due to the huge search space, this is difficult to implement efficiently. Thus, we translate this problem into SAT. Here, we focus on the encoding of POP* for strict precedence with semantic labeling for models over the carrier $\mathbb{B} = \{0,1\}$. To encode a Boolean function $b\colon \mathbb{B}^n \to \mathbb{B}$, we make use of propositional atoms b_w for every sequence of arguments $w = w_1,\ldots,w_n \in \mathbb{B}^n$. Let the formula $\ulcorner b \urcorner(a_1,\ldots,a_n)$ be such that for a satisfying assignment ν, the equality $\nu(\ulcorner b \urcorner(a_1,\ldots,a_n)) = b_{\nu(a_1),\ldots,\nu(a_n)}$ is asserted. For every assignment α and term t appearing in $\mathcal{R}$ we introduce the atoms $\mathrm{int}_{\alpha,t}$ and $\mathrm{lab}_{\alpha,t}$ for $t \notin \mathcal{V}$. The formula $\mathrm{int}_{\alpha,t}$ encodes $[\alpha]_{\mathcal{B}}(t)$, where $[\alpha]_{\mathcal{B}}$ denotes the evaluation function of the Boolean model $\mathcal{B}$, while $\mathrm{lab}_{\alpha,t}$ expresses the label of the root symbol of t with respect to the assignment α. We define $\mathrm{INT}_\alpha(t) = \mathrm{int}_{\alpha,t} \leftrightarrow \ulcorner f_{\mathcal{B}} \urcorner(\mathrm{int}_{\alpha,t_1},\ldots,\mathrm{int}_{\alpha,t_n})$

[1] Available online at `http://cl-informatik.uibk.ac.at/software/icct`, where full evidence of the experiments presented below can be found.
[2] `http://libresource.inria.fr/projects/crocus`
[3] `http://colo6-c703.uibk.ac.at/ttt2/`
[4] `http://minisat.se`

and $\mathrm{LAB}_\alpha(t) = \mathrm{lab}_{\alpha,t} \leftrightarrow \ulcorner \ell_f \urcorner(\mathrm{int}_{\alpha,t_1}, \dots, \mathrm{int}_{\alpha,t_n})$. Further for $t \in \mathcal{V}$ we define $\mathrm{INT}_\alpha(t) = \mathrm{int}_{\alpha,t} \leftrightarrow \alpha(t)$. These constraints have to be enforced for every term appearing in $\mathcal{R}$. We write $\mathcal{R} \trianglerighteq t$ to denote that t is a subterm of a left- or right-hand side of a rule in $\mathcal{R}$. The model condition is formalised by

$$\mathrm{LAB}(\mathcal{R}) = \bigwedge_\alpha \Big(\bigwedge_{\mathcal{R} \trianglerighteq t} (\mathrm{INT}_\alpha(t) \wedge \mathrm{LAB}_\alpha(t)) \wedge \bigwedge_{l \to r \in \mathcal{R}} (\mathrm{int}_{\alpha,l} \leftrightarrow \mathrm{int}_{\alpha,r}) \Big) \ .$$

Assume ν is a satisfying assignment for $\mathrm{LAB}(\mathcal{R})$ and $\mathcal{R}_{\mathsf{lab}}$ denotes the system obtained by labeling $\mathcal{R}$. In order to show compatibility of $\mathcal{R}_{\mathsf{lab}}$ with POP*, we need to find a precedence $>$ and a safe mapping such that $\mathcal{R}_{\mathsf{lab}} \subseteq >_{\mathsf{pop}*}$ holds. To compare the labelled versions of two terms s, t under assignment α, we define

$$\ulcorner s \succ_{\mathsf{pop}*} t \urcorner_\alpha = \ulcorner s >^{(1)}_{\mathsf{pop}*} t \urcorner_\alpha \vee \ulcorner s >^{(2)}_{\mathsf{pop}*} t \urcorner_\alpha \vee \ulcorner s >^{(3)}_{\mathsf{pop}*} t \urcorner_\alpha \vee \ulcorner s >^{(4)}_{\mathsf{pop}*} t \urcorner_\alpha \ .$$

Here $\ulcorner s >^{(i)}_{\mathsf{pop}*} t \urcorner$ refers to the encoding of the case $\langle i \rangle$ from the definition of $\succ_{\mathsf{pop}*}$. We discuss the cases $\langle 2 \rangle$ – $\langle 4 \rangle$ as case $\langle 1 \rangle$ is obtained similarly.

Set $\ulcorner f(s_1, \dots, s_n) >^{(2)}_{\mathsf{pop}*} t \urcorner_\alpha = \top$ if $s_i = t$ holds for some s_i. Otherwise, $\ulcorner f(s_1, \dots, s_n) >^{(2)}_{\mathsf{pop}*} t \urcorner_\alpha = \bigvee_{i=1}^n \ulcorner s_i \succ_{\mathsf{pop}*} t \urcorner_\alpha$. For any $f \in \mathcal{F}_{\mathsf{lab}}$ and an argument position i of f we encode $i \in \mathsf{safe}(f)$ by an atom $\mathsf{safe}_{f,i}$. For $f \in \mathcal{F}$ and a formula a denoting the label of f, the formula $\mathrm{SF}(f_a, i)$ ($\mathrm{NRM}(f_a, i)$) assesses that the i-th position of f_a is safe (normal), respectively. Furthermore, for $f, g \in \mathcal{F}$ and formulae a and b, the formula $\ulcorner f_a > g_b \urcorner$ represents the comparison $f_{\nu(a)} > g_{\nu(b)}$. Assume $f \in \mathcal{D}$. We translate case $\langle 3 \rangle$ as follows:

$$\ulcorner f(s_1, \dots, s_n) >^{(3)}_{\mathsf{pop}*} g(t_1, \dots, t_m) \urcorner_\alpha = \ulcorner f_{\mathrm{lab}_{\alpha,s}} > g_{\mathrm{lab}_{\alpha,t}} \urcorner \wedge$$
$$\bigvee_{i_0} \Big(\ulcorner s \succ_{\mathsf{pop}*} t_{i_0} \urcorner_\alpha \wedge \mathrm{SF}(g_{\mathrm{lab}_{\alpha,t}}, i_0) \wedge \bigwedge_{i \neq i_0} (\ulcorner s >^{(1)}_{\mathsf{pop}*} t_i \urcorner_\alpha \vee (\mathrm{SF}(g_{\mathrm{lab}_{\alpha,t}}, i) \wedge \ulcorner s \rhd t_i \urcorner)) \Big)$$

In order to guarantee that f_a is defined we add a rule $f_a(x_1, \dots, x_n) \to \mathsf{c}$ to the labelled TRS, where c is a fresh constant. In practical tests this has shown to be more effective than allowing that f_a becomes undefined. A careful analysis revealed that in order to encode the restricted multiset comparison, the notion of *multiset covers* [16] is adaptable, cf. [6]. A multiset cover is a pair of total mappings $\gamma \colon \{1, \dots, n\} \to \{1, \dots, n\}$ and $\varepsilon \colon \{1, \dots, n\} \to \mathbb{B}$, encoded using fresh atoms $\gamma_{i,j}$ and ε_i. To assert a correct encoding of (γ, ε), we introduce the formula $\ulcorner (\gamma, \varepsilon) \urcorner$. We define $\ulcorner f(s_1, \dots, s_n) >^{(4)}_{\mathsf{pop}*} f(t_1, \dots, t_n) \urcorner_\alpha$ by

$$(\mathrm{lab}_{\alpha,s} \leftrightarrow \mathrm{lab}_{\alpha,t}) \wedge \bigvee_{i=1}^n (\mathrm{NRM}(f_{\mathrm{lab}_{\alpha,s}}, i) \wedge \neg \varepsilon_i) \wedge \bigwedge_{i=1}^n \bigwedge_{j=1}^n \Big(\gamma_{i,j} \to (\varepsilon_i \to \ulcorner s_i = t_j \urcorner)$$
$$\wedge (\neg \varepsilon_i \to \ulcorner s_i \succ_{\mathsf{pop}*} t_j \urcorner_\alpha) \wedge (\mathrm{SF}(f_{\mathrm{lab}_{\alpha,s}}, i) \leftrightarrow \mathrm{SF}(f_{\mathrm{lab}_{\alpha,t}}, j)) \Big) \wedge \ulcorner (\gamma, \varepsilon) \urcorner \ .$$

Finally $\mathrm{POP}^*_{\mathrm{SL}}(\mathcal{R}) = \bigwedge_\alpha \bigwedge_{l \to r \in \mathcal{R}} \ulcorner l \succ_{\mathsf{pop}*} r \urcorner_\alpha \wedge \mathrm{SM}(\mathcal{R}) \wedge \mathrm{STRICT}(\mathcal{R}) \wedge \mathrm{LAB}(\mathcal{R})$ asserts the existence of a model $\mathcal{B}$ and labeling ℓ such that the labelled TRS is compatible with $\succ_{\mathsf{pop}*}$. Here STRICT($\mathcal{R}$) formalises the strictness of $>$, while SM($\mathcal{R}$) enforces a valid encoding of the safe mapping safe.

4 Experiments and Conclusions

We have tested our complexity tool `icct` on two testbeds: first, we filtered the TPDB for constructor TRS, this testbed is denoted as **C** below. Secondly we considered TRS from the example collections [17,18,19] and filtered for orthogonal constructor (hence confluent) TRS, we call this testbed **FP**.[5] The first test suite (containing 592 systems) aims to compare the relative power of the methods on a decently sized testbed. The second one (containing 71 systems) tries to focus on natural examples of confluent TRS. Success on the testbed **FP** implies that the function encoded by the given TRS is polytime computable — however, for LMPO and POP* one has to verify in addition that the signature is simple. The tests were run on a standard PC with 512 MB of memory and a 2.4 GHz Intel® Pentium™ IV processor. We summarise the results of the test in Table 1.

Counting the total number of yes-instances reveals that the most powerful methods for showing computability of functions in polynomial time are SMC and POP$^*_{\mathrm{SL}}$. As POP$^*_{\mathrm{SL}}$is a mostly syntactic method, it is much faster than the fully semantic SMC. Note that SMC is slightly more powerful than POP$^*_{\mathrm{SL}}$ on testbed **FP**, while the order is reversed for test suite **C**. This indicates that the techniques can handle different TRS, which can be verified by a close look at the full experimental evidence[1]. More precisely, any of the methods implemented in our system `icct` can handle (at least) one example that cannot be handled by any other method. In particular SMC is the only technique that can handle the example `AG01/#3.7`, an encoding of the logarithm function also studied in [8]. In addition we compared `icct` with the results for `crocus` published at the `crocus` site[2], but found that `crocus` in conjunction with PPO [7] cannot handle more TRS than listed in the respective column.

Table 1. Experimental results

	POP*		POP$^*_{\sim}$		POP$^*_{\mathrm{SL}}$		LMPO		RPO^{QI}_{Pro}		SMC	
	FP	**C**	**FP**	**C**	**FP**	**C**	**FP**	**C**	**FP**	**C**	**FP**	**C**
Yes	7	41	8	43	14	74	13	54	13	51	15	69
Maybe	64	551	63	549	57	511	58	538	58	540	49	394
Timeout	0	0	0	0	0	7	0	0	0	1	7	129
Avg. Time (ms)	53		47		192		44		279		507	

As already mentioned, possible future work would be to extend `icct` by implementing sup-interpretations [4,5] and adding the max-function [13] to quasi-interpretations and SMC. In addition we want to investigate whether recent work on context-dependent interpretations [20] can be extended to implicit computational complexity analysis.

[5] We used TPDB version 4.0, available online at http://www.lri.fr/~marche/tpdb/

References

1. Marion, J.Y., Moyen, J.Y.: Efficient first order functional program interpreter with time bound certifications. In: Parigot, M., Voronkov, A. (eds.) LPAR 2000. LNCS (LNAI), vol. 1955, pp. 25–42. Springer, Heidelberg (2000)
2. Bonfante, G., Cichon, A., Marion, J.Y., Touzet, H.: Algorithms with polynomial interpretation termination proof. JFP 11(1), 33–53 (2001)
3. Marion, J.Y.: Analysing the implicit complexity of programs. IC 183, 2–18 (2003)
4. Marion, J.Y., Péchoux, R.: Resource analysis by sup-interpretation. In: Hagiya, M., Wadler, P. (eds.) FLOPS 2006. LNCS, vol. 3945, pp. 163–176. Springer, Heidelberg (2006)
5. Marion, J.Y., Péchoux, R.: Quasi-friendly sup-interpretations. CoRR abs/cs/0608020 (2006)
6. Avanzini, M., Moser, G.: Complexity analysis by rewriting. In: Proc. 9th FLOPS. LNCS, vol. 4989, pp. 130–146. Springer, Heidelberg (2008)
7. Bonfante, G., Marion, J.Y., Moyen, J.Y.: Quasi-intepretations and small space bounds. In: Giesl, J. (ed.) RTA 2005. LNCS, vol. 3467, pp. 150–164. Springer, Heidelberg (2005)
8. Bonfante, G., Marion, J.Y., Péchoux, R.: Quasi-interpretation synthesis by decomposition. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) ICTAC 2007. LNCS, vol. 4711, pp. 410–424. Springer, Heidelberg (2007)
9. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1998)
10. 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)
11. Steinbach, J.: Generating polynomial orderings. IPL 49, 85–93 (1994)
12. Zantema, H.: Termination of term rewriting by semantic labelling. FI 24, 89–105 (1995)
13. Fuhs, C., Giesl, J., Middeldorp, A., Schneider-Kamp, P., Thiemann, R., Zankl, H.: Maximal termination. In: Proc. 19th RTA 2008. LNCS, vol. 5117 (to appear, 2008)
14. Contejean, E., Marché, C., Tomás, A.P., Urbain, X.: Mechanically proving termination using polynomial interpretations. JAR 34(4), 325–363 (2005)
15. 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)
16. Schneider-Kamp, P., Thiemann, R., Annov, E., Codish, M., Giesl, J.: Proving termination using recursive path orders and SAT solving. In: Konev, B., Wolter, F. (eds.) FroCos 2007. LNCS (LNAI), vol. 4720, pp. 267–282. Springer, Heidelberg (2007)
17. Steinbach, J., Kühler, U.: Check your ordering - termination proofs and open problems. Technical Report SR-90-25, University of Kaiserslautern (1990)
18. Dershowitz, N.: 33 examples of termination. In: Term Rewriting, French Spring School of Theoretical Computer Science, Advanced Course, pp. 16–26. Springer, Heidelberg (1995)
19. Arts, T., Giesl, J.: A collection of examples for termination of term rewriting using dependency pairs. Technical Report AIB-09-2001, RWTH Aachen (2001)
20. Moser, G., Schnabl, A.: Proving quadratic derivational complexities using context dependent interpretations. In: Proc. 19th RTA 2008. LNCS, vol. 5117 (to appear, 2008)

LogAnswer - A Deduction-Based Question Answering System (System Description)

Ulrich Furbach[1], Ingo Glöckner[2], Hermann Helbig[2], and Björn Pelzer[1]

[1] Department of Computer Science, Artificial Intelligence Research Group
University of Koblenz-Landau, Universitätsstr. 1, 56070 Koblenz
{uli,bpelzer}@uni-koblenz.de

[2] Intelligent Information and Communication Systems Group (IICS),
University of Hagen, 59084 Hagen, Germany
{ingo.gloeckner,hermann.helbig}@fernuni-hagen.de

Abstract. LogAnswer is an open domain question answering system which employs an automated theorem prover to infer correct replies to natural language questions. For this purpose LogAnswer operates on a large axiom set in first-order logic, representing a formalized semantic network acquired from extensive textual knowledge bases. The logic-based approach allows the formalization of semantics and background knowledge, which play a vital role in deriving answers. We present the functional LogAnswer prototype, which consists of automated theorem provers for logical answer derivation as well as an environment for deep linguistic processing.[1]

1 Introduction

Question answering (QA) systems generate natural language (NL) answers in response to NL questions, using a large collection of textual documents. Simple factual questions can be answered using only information retrieval and shallow linguistic methods like named entity recognition. More advanced cases, like questions involving a temporal description, call for deduction based question answering which can provide support for temporal reasoning and other natural language related inferences. Logic has been used for such semantic NL analysis in the DORIS [1] system, although this is aimed at discourse regarding a limited domain instead of open-domain QA. There are also several examples of logic-based QA systems (like PowerAnswer [2] and Senso [3]), as well as dedicated components for logical answer validation like COGEX [4] or MAVE [5]. However, most of these solutions are research prototypes developed for the TREC or CLEF evaluation campaigns, which ignore the issue of processing time.[2] For actual users, getting the answer in a few seconds is critical to the usefulness of a QA system, though. A QA system must achieve these response times with

[1] Funding of this work by the DFG (Deutsche Forschungsgemeinschaft) under contracts FU 263/12-1 and HE 2847/10-1 (LogAnswer) is gratefully acknowledged.

[2] See http://trec.nist.gov/ (TREC), http://www.clef-campaign.org/ (CLEF).

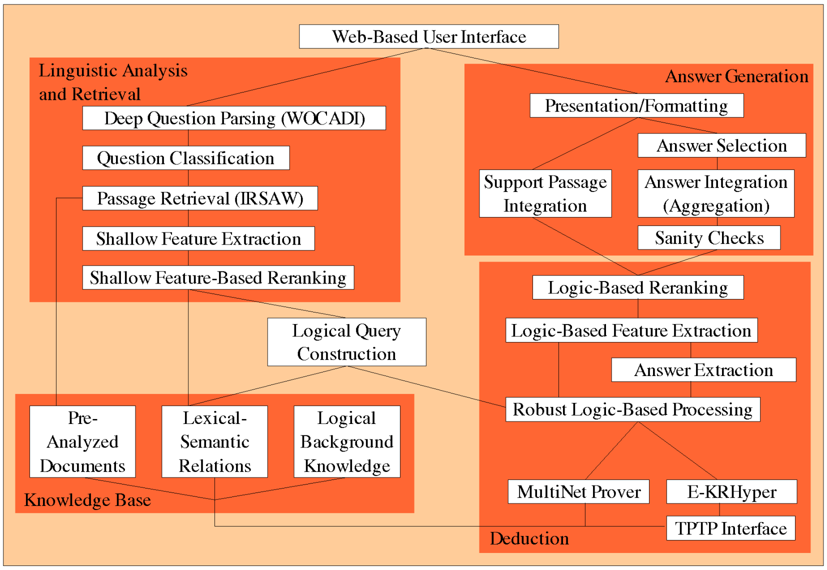

Web-Based User Interface
Linguistic Analysis and Retrieval
Deep Question Parsing (WOCADI)
Question Classification
Passage Retrieval (IRSAW)
Shallow Feature Extraction
Shallow Feature-Based Reranking
Answer Generation
Presentation/Formatting
Answer Selection
Support Passage Integration
Answer Integration (Aggregation)
Sanity Checks
Logic-Based Reranking
Logic-Based Feature Extraction
Answer Extraction
Logical Query Construction
Robust Logic-Based Processing
Pre-Analyzed Documents
Lexical-Semantic Relations
Logical Background Knowledge
Knowledge Base
MultiNet Prover
E-KRHyper
TPTP Interface
Deduction

Fig. 1. System architecture of the LogAnswer prototype

a knowledge base generated from tens of millions of sentences. A second challenge for logic-based QA is robustness. The processing chain of a logic-based QA system involves many stages (from NL via syntactic-semantic parsing to logical forms and knowledge processing and back to NL answers). Therefore fallback solutions like the usage of shallow features are needed to ensure baseline performance when one of the deep NLP modules fails, and gaps in the background knowledge must be bridged by robustness-enhancing techniques like relaxation.

2 Description of the LogAnswer System

The system architecture of the LogAnswer QA system is shown in Fig. 1. In the following we describe the processing stages of the system.

User interface. The natural language question is entered into the LogAnswer web search box.[3] Depending on user preferences, the system answers the question by presenting supporting text passages only or alternatively, by presenting exact answers together with the supporting passage.

Deep Question Parsing. The question is analyzed by the WOCADI parser [6], which generates a semantic representation of the question in the MultiNet formalism [7]. A question classification is also done in this phase, which currently discerns only definition questions (*What is a neutrino?*) and factual questions

[3] The system is available online at `www.loganswer.de`.

(*Who discovered the neutrino?*). While factual questions can be answered by logical means alone, definition questions need additional filtering in order to identify descriptions that are not only true but also represent defining knowledge.

Passage Retrieval. The document collection of LogAnswer comprises the CLEF news collection and a snapshot of the German Wikipedia (17 million sentences total). In order to avoid parsing of documents at query time, all documents are pre-analyzed by the WOCADI parser. The resulting MultiNet representations are segmented into passages and stored in the IRSAW retrieval module [8], which uses the terms in the passage for indexing.[4] Given the query terms, IRSAW typically retrieves 200 (or more) passages as the basis for logical answer finding.

Shallow Feature Extraction and Reranking. In order to avoid logical processing of all retrieved passages, LogAnswer tries to identify the most promising cases by reranking passages using shallow features (like overlap of lexical concepts, proper names and numerals of the question with those found in the passage). It is important that these features can be computed very quickly without the help of the prover. The machine learning approach and the set of shallow features are detailed in [9,10].

Logical Query Construction. The semantic network for the question is turned into a conjunctive list of query literals. Synonyms are normalized by replacing all lexical concepts with canonical synset representatives.[5] For example, *Wie viele Menschen starben beim Untergang der Estonia?*[6] translates into the following logical query (with the *FOCUS* variable representing the queried information):

$$\begin{aligned}&\mathrm{sub}(X_1, \mathit{estonia.1.1}), \mathrm{attch}(X_1, X_2), \mathrm{subs}(X_2, \mathit{untergang.1.1}), \mathrm{subs}(X_3, \mathit{sterben.1.1}),\\&\mathrm{circ}(X_3, X_2), \mathrm{aff}(X_3, FOCUS), \mathrm{pred}(FOCUS, \mathit{mensch.1.1})\,.\end{aligned}$$

Robust Logic-Based Processing. As the basis for answer extraction and for improving the passage ranking, LogAnswer tries to prove the logical representation of the question from the representation of the passage and the background knowledge.[7] Robustness is gained by using relaxation: if a proof is not found within a time limit, then query literals are skipped until a proof of the remaining query succeeds, and the skip count indicates (non-)entailment [5,10]. For efficiency reasons, relaxation is stopped before all literals are proved or skipped. One can then state upper/lower bounds on the provable literal count, assuming that all (or none) of the remaining literals are provable.

[4] The current version of LogAnswer uses a segmentation into single sentences, but we will also experiment with different passage sizes (paragraphs and full documents).

[5] The system uses 48,991 synsets (synonym sets) for 111,436 lexical constants.

[6] *How many people died when the MS Estonia sank?*

[7] The background knowledge of LogAnswer comprises 10,000 lexical-semantic facts (e.g. for nominalizations) and 109 logical rules, which define main characteristics of MultiNet relations and also handle meta verbs like 'stattfinden' (take place) [5].

Answer Extraction. If a proof of the question from a passage succeeds, then LogAnswer obtains an answer binding which represents the queried information. For finding more answers, the provers of LogAnswer can also return a substitution for a proven query fragment when a full proof fails. Given a binding for the queried variable, LogAnswer uses word alignment hints of WOCADI for finding the matching answer string, which is directly cut from the original text passage.

Logic-Based Feature Extraction. For a logic-based refinement of relevance scores, LogAnswer extracts the following features, which depend on the limit on relaxation cycles and on the results of answer extraction:

- *skippedLitsLb* Number of literals skipped in the relaxation proof.
- *skippedLitsUb* Number of skipped literals, plus literals with unknown status.
- *litRatioLb* Relative proportion of actually proved literals compared to the total number of query literals, i.e. $1 - \mathit{skippedLitsUb}/\mathit{allLits}$.
- *litRatioUb* Relative proportion of potentially provable literals (not yet skipped) vs. all query literals, i.e. $1 - \mathit{skippedLitsLb}/\mathit{allLits}$.
- *boundFocus* Indicates that a binding for the queried variable was found.
- *npFocus* Indicates that the queried variable was bound to a constant which corresponds to a nominal phrase (NP) in the text.
- *phraseFocus* Signals that an answer string has been extracted.

Logic-Based Reranking. The logic-based reranking of the passages uses the same ML approach as the shallow reranking, but the shallow and logic-based features are now combined for better precision. Rather than computing a full reranking, passages are considered in the order determined by the shallow feature-based ranking, and logical processing is stopped after a pre-defined time limit.

Support Passage Selection. When using LogAnswer for retrieving text snippets which contain an answer, all passages are re-ranked using either the logic-based score (if available for the passage) or the shallow-feature score (if there is no logic-based result for the passage due to parsing failure or time restrictions). The top k passages are chosen for presentation ($k = 5$ for the web interface).

Sanity Checks. When the user requests exact answers rather than snippets which contain the answer, additional processing is needed: a triviality check eliminates answers which only repeat contents of the question. For the question *Who is Virginia Kelley?*, this test rejects trivial answers like *Virginia* or *Virginia Kelley*. A special sanity check for definition questions also rejects the non-informative answer *the mother* (instead of the expected *the mother of Bill Clinton*), see [5].

Aggregation and Answer Selection. The answer integration module computes a global score for each answer, based on the local score for each passage from which the answer was extracted. The aggregation method already proved effective in [5]. The $k = 5$ distinct answers with the highest aggregated scores are then selected for presentation. For each answer, the supporting passage with the highest score is also shown in order to provide a justification for the presented answer.

3 Theorem Provers of LogAnswer

The robust logic-based processing (see Section 2) has to merge contrasting goals: it has to derive answers from a logical knowledge representation using precise inference methods, but it must also provide these answers within acceptable response times and account for imperfections of the textual knowledge sources and their formalization. Thus a theorem prover must meet several requirements if it is to serve as the deduction component in LogAnswer.

Handling of Large Knowledge Bases. Of high importance is the ability to work on the large set of axioms and facts forming the knowledge base, which will keep growing in the future. This includes the way these clauses are supplied to the prover. Theorem provers usually operate on a single-problem basis: the prover is started with the required clauses and then terminates after a successful proof derivation. For LogAnswer this approach is impractical. In order to achieve a usability comparable to conventional search engines, the system should spend all of the available processing time for actual reasoning and not for loading the background knowledge into the prover. Since any two query tasks use the same background knowledge and only differ in a few clauses representing the query and a text passage, a LogAnswer prover should be able to stay in operation to perform multiple query tasks, loading and retracting the query-specific clauses while keeping the general knowledge base in the system.

Relaxation Loop Support. The prover must also support the robustness enhancing techniques, in particular by providing guidance to the relaxation loop. The large knowledge base with its imperfections often causes the prover to reach the time limit, where LogAnswer will interrupt the reasoning and relax the query. The prover must then report details about its failed proof attempt so that the relaxation loop can select the query literal most suited for skipping.

Answer Extraction Support. Finally, if a proof succeeds, then the prover must state any answer substitutions found for the *FOCUS* variable.

The current LogAnswer prototype includes two theorem provers for comparison purposes in the development phase.

The MultiNet Prover. The prover of the MultiNet toolset[8] is based on SLD resolution and operates on range-restricted Horn formulas. While very limited in expressive power, it can prove a question from a passage in less than 20ms on average [9]. The prover was optimized by using term indexing, caching, lazy indexing, optimizing literal ordering, and by using profiling tools.

The E-KRHyper Prover. The other system is E-KRHyper [11], a theorem prover for full first order logic with equality, including input which is not Horn and not range restricted. Currently existing alongside the MultiNet prover,

[8] See `http://pi7.fernuni-hagen.de/research/mwrplus`

E-KRHyper will eventually become the sole reasoning component of LogAnswer once a new translation of MultiNet representations into full first-order logic has been completed. E-KRHyper implements the E-hyper tableau calculus [12]. Designed for use as an embedded knowledge processing engine, the system has been employed in a number of knowledge representation applications. It is capable of handling large sets of uniformly structured input facts, and it can provide proof output for models and refutations. Input is accepted in TPTP syntax [13]. E-KRHyper features several extensions to first-order logic, like arithmetic evaluation, negation as failure and builtin predicates adapted from Prolog. These will be helpful in the ongoing translation of the knowledge base into logic, allowing us to capture the full expressivity of the MultiNet formalism. Compared to other full first-order theorem provers, E-KRHyper also has the pragmatic advantage of being an in-house system, easily tailored to any upcoming difficulties, instead of a black box which we must adapt to.

In the LogAnswer system E-KRHyper is embedded as a reasoning server, and thus it remains in constant operation. On its startup E-KRHyper is supplied with MultiNet background knowledge translated into first-order TPTP syntax. The prover further transforms this into clause normal form, a requirement for the tableaux-based reasoning algorithm of E-KRHyper. Currently this CNF-representation consists of approximately 10,000 clauses. Discrimination-tree indexing serves to maintain this clause set efficiently. Loading the knowledge base into E-KRHyper requires circa four seconds on our test bed system.[9]

Given that an answer to the query may be found in any of the supporting passages (see Section 2), E-KRHyper runs an independent proof attempt for each passage. For such an attempt the query clause (consisting of the negated query literals) and the logical passage representation are added to E-KRHyper's set of clauses. The average query clause for a question from the CLEF-07 contains eight literals, and the average translated passage is a set of 230 facts.

E-KRHyper then tries to find a refutation for the given input. The main LogAnswer system is notified if the prover succeeds. Also, if specific information using a *FOCUS* variable is requested (as described before), then the binding of this variable is retrieved from the refutation and returned to the answer extractor. Finally, E-KRHyper drops all clauses apart from the background knowledge axioms and is ready for the next query or passage.

If on the other hand E-KRHyper fails to find a refutation within the time limit, then it halts the derivation and provides *relaxation loop support* by delivering partial results. These represent the partly successful refutation attempts made so far: during the derivation E-KRHyper evaluates the query clause from left to right, trying to unify all literals with complementary unit clauses from the current tableau branch and thereby yielding a refutation. If a literal cannot be unified, the remaining unevaluated query literals are not considered and this attempt stops. Each partial result represents such a failed evaluation; it consists of the subset of refuted query literals, the unifying substitutions and the failed query literal. LogAnswer selects one of the 'best' partial results (i.e.

[9] Intel Q6600 2.4 GHz

where most query literals could be refuted) and removes the failed literal from the query. E-KRHyper resets its clause set to the background knowledge axioms, and the relaxation loop restarts the derivation with the shortened query. This process is repeated, with another query literal being skipped in each round, until E-KRHyper derives a refutation for the current query fragment or the bound for the number of skipped query literals is reached, see Section 2.[10]

Addressing the *handling of large knowledge bases*, the methods described above reset the clause input before every new task. This is facilitated by the prover's ability to save and restore states of its knowledge base. That way the prover can rapidly drop obsolete subsets of the clauses and their discrimination-trees, with no need to rebuild the extensive index for the background axioms.

To estimate the performance of the prover for our intended use we tested E-KRHyper without relaxation on 1806 query/passage-combinations from CLEF-07 which are known to contain answers. 77 (4.3%) of these could not be proven within a 360 seconds time limit set for each test case, in part due to the yet incomplete logical translation of the MultiNet background knowledge. In the remaining cases E-KRHyper required on average 1.97 seconds for each proof. 895 proofs (49.6%) could be found in less than one second, and the maximum time needed was 37 seconds. This illustrates that relaxation, and the extraction of answer bindings from incomplete proofs, are imperative when processing time is critical (as in ad-hoc question answering on the web), and multiple passages must be processed for a single query in a very short time frame. However, it also shows that more precise answers can be found within a time frame which may still be acceptable for specific applications where time is less important than quality, so our approach is easily scaled to different uses.

4 Conclusions and Future Work

We have presented a logic-based question answering system which combines an optimized deductive subsystem with shallow techniques by machine learning. The prototype of the LogAnswer system, which can be tested on the web, demonstrates that the response times achieved in this way are suitable for ad-hoc querying. The quality of passage reranking has been measured for factual questions from CLEF-07: On the retrieved passages, the ML classifier, which combines deep and shallow features, obtains a filtering precision of 54.8% and recall of 44.8% [10]. In the future, the potential of E-KRHyper will be exploited by formalizing more expressive axioms that utilize equality and non-Horn formulas. The capability of LogAnswer to find exact answers (rather than support passages) will be assessed in the CLEF-08 evaluation.

[10] Continuing our example from Section 2, this is one of the candidate passages that will be found and analysed: *(...) the sinking of the ferry 'Estonia' (...) cost the most human lives: over 900 people died (...)* . After three relaxation steps a proof is found with the value *900* bound to the *FOCUS* variable, and LogAnswer returns the answer *over 900 people* to the user.

References

1. Bos, J.: Doris 2001: Underspecification, resolution and inference for discourse representation structures. In: ICoS-3 - Inference in Compuational Semantics, Workshop Proceedings (2001)
2. Moldovan, D., Bowden, M., Tatu, M.: A temporally-enhanced PowerAnswer in TREC 2006. In: Proc. of TREC 2006, Gaithersburg, MD (2006)
3. Saias, J., Quaresma, P.: The Senso question answering approach to Portuguese QA@CLEF-2007. In: Working Notes for the CLEF 2007 Workshop, Budapest, Hungary (2007)
4. Tatu, M., Iles, B., Moldovan, D.: Automatic answer validation using COGEX. In: Peters, C., Clough, P., Gey, F.C., Karlgren, J., Magnini, B., Oard, D.W., de Rijke, M., Stempfhuber, M. (eds.) CLEF 2006. LNCS, vol. 4730. Springer, Heidelberg (2007)
5. Glöckner, I.: University of Hagen at QA@CLEF 2007: Answer validation exercise. In: Working Notes for the CLEF 2007 Workshop, Budapest (2007)
6. Hartrumpf, S.: Hybrid Disambiguation in Natural Language Analysis. Der Andere Verlag, Osnabrück, Germany (2003)
7. Helbig, H.: Knowledge Representation and the Semantics of Natural Language. Springer, Heidelberg (2006)
8. Leveling, J.: IRSAW – towards semantic annotation of documents for question answering. In: CNI Spring 2007 Task Force Meeting, Phoenix, Arizona (2007)
9. Glöckner, I.: Towards logic-based question answering under time constraints. In: Proc.of ICAIA 2008, Hong Kong, pp. 13–18 (2008)
10. Glöckner, I., Pelzer, B.: Exploring robustness enhancements for logic-based passage filtering. In: Proceedings of KES 2008. LNCS. Springer, Heidelberg (to appear, 2008)
11. Pelzer, B., Wernhard, C.: System Description: E-KRHyper. In: Automated Deduction - CADE-21, Proceedings, pp. 508–513 (2007)
12. Baumgartner, P., Furbach, U., Pelzer, B.: Hyper Tableaux with Equality. In: Automated Deduction - CADE-21, Proceedings, pp. 492–507 (2007)
13. Sutcliffe, G., Suttner, C.: The TPTP Problem Library: CNF Release v1.2.1. Journal of Automated Reasoning 21(2), 177–203 (1998)

A High-Level Implementation of a System for Automated Reasoning with Default Rules (System Description)

Christoph Beierle[1], Gabriele Kern-Isberner[2], and Nicole Koch[1]

[1] Dept. of Computer Science, FernUniversität in Hagen, 58084 Hagen, Germany
[2] Dept. of Computer Science, TU Dortmund, 44221 Dortmund, Germany

Abstract. An overview of a system modelling an intelligent agent being able to reason by using default rules is given. The semantics of the qualitative default rules is defined via ordinal conditional functions which model the epistemic state of the agent, providing her with the basic equipment to perform different knowledge management and belief revision tasks. Using the concept of Abstract State Machines, the fully operational system was developed in AsmL, allowing for a high-level implementation that minimizes the gap between the mathematical specification of the underlying concepts and the executable code in the implemented system.

1 Introduction

When studying concepts and methods for nonmonotonic reasoning, actually implemented and operational systems realizing the developed approaches can be very helpful. In this paper, we give a brief description of the Condor@AsmL system that implements automated reasoning with qualitative default rules [1,6]. It provides functionalities for advanced knowledge management tasks like belief revision and update or diagnosis and hypothetical what-if-analysis. Condor@AsmL implements the abstract Condor specification given in [3] and was developed in AsmL [2,7], allowing for a high-level implementation that minimizes the gap between the mathematical specification of the underlying concepts and the executable code in the implemented system.

2 Background

We start with a propositional language $\mathcal{L}$, generated by a finite set Σ of atoms $a, b, c, \ldots$. The formulas of $\mathcal{L}$ are denoted by uppercase Roman letters $A, B, C, \ldots$. For conciseness of notation, we will omit the logical *and*-connective, writing AB instead of $A \wedge B$, and overlining formulas will indicate negation, i.e. $\overline{A}$ means $\neg A$. Let Ω denote the set of possible worlds over $\mathcal{L}$; Ω will be taken here simply

The research reported here was partially supported by the Deutsche Forschungsgemeinschaft (grants BE 1700/7-1 and KE 1413/2-1).

as the set of all propositional interpretations over $\mathcal{L}$ and can be identified with the set of all complete conjunctions over Σ. For $\omega \in \Omega$, $\omega \models A$ means that the propositional formula $A \in \mathcal{L}$ holds in the possible world ω.

By introducing a new binary operator $|$, we obtain the set $(\mathcal{L} \mid \mathcal{L}) = \{(B|A) \mid A, B \in \mathcal{L}\}$ of conditionals over $\mathcal{L}$. $(B|A)$ formalizes "*if A then B*" and establishes a plausible, probable, possible etc connection between the *antecedent* A and the *consequent* B. Here, conditionals are supposed not to be nested, that is, antecedent and consequent of a conditional will be propositional formulas.

A conditional $(B|A)$ is an object of a three-valued nature, partitioning the set of worlds Ω in three parts: those worlds satisfying AB, thus *verifying* the conditional, those worlds satisfying $A\overline{B}$, thus *falsifying* the conditional, and those worlds not fulfilling the premise A and so which the conditional may not be applied to at all. This allows us to represent $(B|A)$ as a *generalized indicator function* going back to [5] (where u stands for *unknown* or *indeterminate*):

$$(B|A)(\omega) = \begin{cases} 1 & \text{if } \ \omega \models AB \\ 0 & \text{if } \ \omega \models A\overline{B} \\ u & \text{if } \ \omega \models \overline{A} \end{cases}$$

To give appropriate semantics to conditionals, they are usually considered within richer structures such as *epistemic states*. Besides certain (logical) knowledge, epistemic states also allow the representation of, e.g., preferences, beliefs, assumptions of an intelligent agent. Basically, an epistemic state allows one to compare formulas or worlds with respect to plausibility, possibility, necessity, probability, etc.

Well-known qualitative, ordinal approaches to represent epistemic states are Spohn's *ordinal conditional functions, OCFs*, (also called *ranking functions*) [9], and *possibility distributions* [4], assigning degrees of plausibility, or of possibility, respectively, to formulas and possible worlds. In such qualitative frameworks, a conditional $(B|A)$ is valid (or *accepted*), if its confirmation, AB, is more plausible, possible, etc. than its refutation, $A\overline{B}$; a suitable degree of acceptance is calculated from the degrees associated with AB and $A\overline{B}$.

The concept underlying Condor@AsmL is based on Spohn's OCFs [9]. An OCF $\kappa : \Omega \rightarrow \mathbb{N}$ expresses degrees of implausibility of worlds: The higher $\kappa(\omega)$, the less plausible is ω. At least one world must be regarded as being completely normal; therefore, $\kappa(\omega) = 0$ for at least one $\omega \in \Omega$. Such a ranking function can be taken as the representation of a full epistemic state of an agent. Each such κ uniquely extends to a function (also denoted by κ) mapping sentences and rules to $\mathbb{N} \cup \{\infty\}$ defined by

$$\kappa(A) = \begin{cases} min\{\kappa(\omega) \mid \omega \models A\} & \text{if } A \text{ is satisfiable} \\ \infty & \text{otherwise} \end{cases}$$

$$\kappa((B|A)) = \begin{cases} \kappa(AB) - \kappa(A) & \text{if } \kappa(A) \neq \infty \\ \infty & \text{otherwise} \end{cases}$$

for sentences A and conditionals $(B|A)$, respectively.

The beliefs of an agent being in epistemic state κ with respect to a set of default rules $\mathcal{S}$ is determined by

$$belief_{\mathcal{O}}(\kappa, \mathcal{S}) \;=\; \{S[m] \mid S \in \mathcal{S} \text{ and } \kappa \models_{\mathcal{O}} S[m] \text{ and } \kappa \not\models_{\mathcal{O}} S[m+1]\}$$

where the satisfaction relation $\models_{\mathcal{O}}$ for quantified sentences $(B|A)[m]$ is defined by

$$\kappa \models_{\mathcal{O}} (B|A)[m] \quad \text{iff} \quad \kappa(AB) + m \;<\; \kappa(A\overline{B}) \tag{1}$$

Thus, $(B|A)$ is believed in κ with degree of belief m if the degree of disbelief of AB (verifying the unquantified conditional) is more than m smaller than the degree of disbelief of $A\overline{B}$ (falsifying the unquantified conditional).

3 Examples and System Walk-Through

After starting the Condor@AsmL system, the user can choose among its top-level functionalities as depicted in Fig. 1. Since our focus was on a lean implementation, currently there is only a command line and file I/O interface. This enables both an easy integration of Condor@AsmL with other systems and also adding a graphical user interface.

```
****************************
** Experience Condor@AsmL **
** Vers. 2.2 (2008-02-17) **
****************************

Choose an  action:
  [0] Initialization
  [1] Load
  [2] Query
  [3] Belief and Disbelief
  [4] Revision and Update
  [5] Iterated Revision
  [6] Iterated Update
  [7] Diagnosis
  [8] What-if-Analysis
  [9] Exit Program
```

Fig. 1. Start menu

If the user chooses "0" for initializing the system, Condor@AsmL asks for a file name from which the set of rules representing the initial knowledge base should be read. For instance, after reading the file containing the four rules representing the conditionals *$\underline{s}$ea animals have $\underline{g}$ills* $(g|s)$, *$\underline{s}$ea animals are not $\underline{m}$ammals* $(\overline{m}|s)$, *$\underline{f}$lippers are $\underline{s}$ea animals* $(s|f)$, *$\underline{f}$lippers are $\underline{m}$ammals*) $(m|f)$, Condor@AsmL determines the epistemic state obtained from revising the *a priori* state of complete ignorance by the given rules and makes this the current epistemic state (Fig. 2(a)).

Asking queries is done by choosing "2" in the main menu and providing a set Q of (unquantified) rules. Condor@AsmL then infers and prints out the degrees of belief for those rules the agent believes in with respect to its current epistemic state. In the example given in Fig. 2(b), for all three queries the degree 0 is returned. Note that for instance $(s|f)[0]$ indeed expresses the agent's basic belief in flippers being sea animals since $\kappa \models_{\mathcal{O}} (s|f)[0]$ says that the agent considers fs to be strictly more plausible than $f\overline{s}$ (cf. (1)).

Diagnosis requires a set F of given facts and a set D of possible diagnoses (also in the form of simple atoms). For these, Condor@AsmL infers and prints out the degrees of belief for the elements of D with respect to the epistemic state obtained by updating the current epistemic state by the facts F (i.e., by focussing on F). Note that the agent's current epistemic state is not changed.

What-if-Analysis realizes hypothetical reasoning and can be viewed upon as a generalization of diagnostic reasoning: Instead of the set F of simple facts we consider a set A of general rules as assumptions, and instead of the simple atoms as in the set D we consider a set H of general rules as hypotheses whose degrees of belief we are interested in. Thus, given rule sets A and H, Condor@AsmL answers the query: *What would be the degrees of belief for the hypotheses H if the world was such that the assumptions A would hold?* This is achieved by inferring the degrees of belief for the hypotheses H by focussing on A (i.e. with respect to the epistemic state obtained from updating the current epistemic state by the assumptions A). Note that in the example shown in Fig. 2(c), under the given assumption $(-g|f)$ the agent would give up the belief in $(s|f)$. Since neither the goal $(-s|f)$ would be believed, the assumption that flippers do not have gills would cause the agent to have reasonable doubts about flippers being a sea animals and to consider the opposite (flippers not being a sea animal) to be equally plausible.

As with diagnosis, also for What-if-Analysis the agent's current epistemic state is not changed. This is the case for the belief revision operators ("4", "5", "6" in Fig. 1). For instance, the update operation transforms the current epistemic state and a set of default rules $\mathcal{R}$ to a new current epistemic state accepting $\mathcal{R}$. Condor@AsmL respects and implements subtle differences between update and (genuine) revision (cf. [1,8]).

4 Implementation

In order to demonstrate the flavor of the implementation in AsmL [2,7] and the close correspondence between the mathematical specifications on the one hand and the operational code on the other hand, we will consider the notion of c-revision [8] which is at the heart of Condor@AsmL's belief revision and reasoning facilties.

A c-revision transforms an epistemic state and a set of quantified sentences into a new epistemic state accepting these sentences. A characterization theorem of [8] shows that every c-revision $\kappa * \mathcal{R}$ of an epistemic state κ and a set of rules $\mathcal{R}$ can be obtained by adding to each $\kappa(\omega)$ values for each rule $R_i \in \mathcal{R}$, depending on whether ω verifies or falsifies R_i. We will now briefly sketch the approach of [8] how to calculate such a c-revision for any finite OCF κ and any finite consistent set $\mathcal{R}$ of conditionals.

The consistency of a set $\mathcal{R} = \{(B_1|A_1), \ldots, (B_n|A_n)\}$ of conditionals in a qualitative framework can be characterized by the notion of *tolerance*. A conditional $(B|A)$ is said to be tolerated by a set of conditionals $\mathcal{R}$ iff there is a world ω such that ω verifies $(B|A)$ (i.e. $(B|A)(\omega) = 1$) and ω does not falsify any of the conditionals in $\mathcal{R}$ (i.e. $r(\omega) \neq 0$ for all $r \in \mathcal{R}$). $\mathcal{R}$ is consistent iff there is an ordered partition $\mathcal{R}_0, \mathcal{R}_1, \ldots, \mathcal{R}_k$ of $\mathcal{R}$ such that each conditional in $\mathcal{R}_m$ is tolerated by $\bigcup_{j=m}^{k} \mathcal{R}_j$, $0 \leqslant m \leqslant k$ (cf. [6]).

a)
```
RULE_BASE loaded:
  (g|s)
  (-m|s)
  (s|f)
  (m|f)

Current epistemic state:
  [0]:   ( s, g, m, f)  :  1
  [1]:   ( s, g, m,-f)  :  1
  [2]:   ( s, g,-m, f)  :  2
  [3]:   ( s, g,-m,-f)  :  0
  [4]:   ( s,-g, m, f)  :  2
  [5]:   ( s,-g, m,-f)  :  2
  [6]:   ( s,-g,-m, f)  :  3
  [7]:   ( s,-g,-m,-f)  :  1
  [8]:   (-s, g, m, f)  :  2
  [9]:   (-s, g, m,-f)  :  0
 [10]:   (-s, g,-m, f)  :  4
 [11]:   (-s, g,-m,-f)  :  0
 [12]:   (-s,-g, m, f)  :  2
 [13]:   (-s,-g, m,-f)  :  0
 [14]:   (-s,-g,-m, f)  :  4
 [15]:   (-s,-g,-m,-f)  :  0
```

b)
```
QUERIES loaded:
  (m|f)
  (g|f)
  (s|f)

Believed sentences:
    [2]:  (m|f)[0]
    [1]:  (g|f)[0]
    [0]:  (s|f)[0]
```

c)
```
ASSUMPTIONS loaded:
  (-g|f)

GOALS loaded:
  (m|f)
  (-g|f)
  (s|f)
  (-s|f)

Focussed epistemic state:
  [0]:  ( s, g, m, f)  :  4
  [1]:  ( s, g, m,-f)  :  1
  [2]:  ( s, g,-m, f)  :  5
  [3]:  ( s, g,-m,-f)  :  0
  [4]:  ( s,-g, m, f)  :  2
  [5]:  ( s,-g, m,-f)  :  2
  [6]:  ( s,-g,-m, f)  :  3
  [7]:  ( s,-g,-m,-f)  :  1
  [8]:  (-s, g, m, f)  :  5
  [9]:  (-s, g, m,-f)  :  0
 [10]:  (-s, g,-m, f)  :  7
 [11]:  (-s, g,-m,-f)  :  0
 [12]:  (-s,-g, m, f)  :  2
 [13]:  (-s,-g, m,-f)  :  0
 [14]:  (-s,-g,-m, f)  :  4
 [15]:  (-s,-g,-m,-f)  :  0

Believed sentences:
    [0]:  (m|f)[0]
    [1]:  (-g|f)[1]
```

Fig. 2. Initialization (a), Query (b) and What-If-Analysis (c)

The corresponding AsmL code for inferring the consistency of a set of rules is given by

```
isConsistent(in_R as RuleSet) as Boolean
  step Partition := buildPartition(in_R, {->})
  step return Partition <> {->}  // consistent iff Partition is non-empty
```

where *buildPartition(in_R,p)* is a binary recursive function returning an ordered partition (i.e. a function mapping natural numbers to sets of rules) of *in_R* if it exists, and the empty map otherwise. *buildPartition(in_R,p)* takes a set of rules *in_R* (still to be partitioned) and a partition p (of those rules that have already been assigned to a paricular partition set R_m). Initially, *in_R* contains all given rules and p is the empty function {->}. The definiton of *buildPartition* is given in Fig. 3.

```
buildPartition(in_R as RuleSet, in_partition as Map of Integer to RuleSet)
                                              as Map of Integer to RuleSet
// recursively build proper partition
 var rules as RuleSet = in_R
 var partition as Map of Integer to RuleSet = in_partition
 let tolerating_worlds = {w | w in Omega where falsify(w,rules) = {} }
 let tolerated_rules = {r | w in tolerating_worlds, r in verify(w,rules)}
 step
  if tolerated_rules = {}
  then partition := {->}   // proper partition does not exist
  else                     // extend current partition
    let next_index = Size(partition)   // next index, starts with 0
    step partition := partition + {next_index -> tolerated_rules}
         rules := rules - tolerated_rules
    step if rules <> {}  // partition remaining rules recursively
         then partition := buildPartition(rules, partition)
 step return partition
```

Fig. 3. AsmL code for determining an ordered partition of a set of rules

```
cRevision(in_kappa as State, in_R as RuleSet) as State
// in_R must be consistent, Partition must contain its ordered partition
  var kappa as State = {->}
  step
    if accept(in_kappa, in_R) = in_R    // all rules accepted
    then kappa := in_kappa              // no change of in_kappa needed
    else
      let kappaMinus = getKappaMinus(in_kappa)
      kappa := {w -> k + sumPenalty(w, in_R, kappaMinus) | w -> k in
                                                              in_kappa}
      let kappaNull = min i | world -> i in kappa
      if kappaNull > 0                      // normalization required?
      then kappa := {w -> k - kappaNull | w -> k in kappa}
  step return kappa
```

Fig. 4. Computation of a cRevision of an epistemic state with respect to a set of rules

Now suppose that $\mathcal{R}$ is consistent and that a corresponding partition $\mathcal{R}_0, \mathcal{R}_1, \ldots, \mathcal{R}_k$ of $\mathcal{R}$ is given. Then the following yields a c-revision: Set successively, for each partitioning set $\mathcal{R}_m$, $0 \leqslant m \leqslant k$, starting with $\mathcal{R}_0$, and for each conditional $r_i = (B_i|A_i) \in \mathcal{R}_m$

$$\kappa_i^- \;:=\; 1 \;+\; \min_{\substack{\omega \models A_i B_i \\ r(\omega) \neq 0, \forall r \in \mathcal{R}_m \cup \ldots \cup \mathcal{R}_k}} \left(\kappa(\omega) \;+\; \sum_{\substack{r_j \in \mathcal{R}_0 \cup \ldots \cup \mathcal{R}_{m-1} \\ r_j(\omega) = 0}} \kappa_j^-\right) \tag{2}$$

Finally, choose κ_0 appropriately to make

$$\kappa^*(\omega) \;=\; \kappa_0 + \kappa(\omega) + \sum_{\substack{1 \leqslant i \leqslant n \\ \omega \models A_i \overline{B}_i}} \kappa_i^- \tag{3}$$

an ordinal conditional function.

The binary function *cRevision(in_kappa,in_R)* as depicted in Fig. 4 takes an epistemic state *in_kappa* and a consistent set of rules *in_R* (where the global nullary function *Partition* already denotes an ordered partition of *in_R*) and returns a correspondingly updated state. If all rules in *in_R* are already accepted by *in_kappa*, then no change is needed and the given *in_kappa* is returned. Otherwise, according to equation 2, *getKappaMinus(in_kappa)* determines the κ_i^- value for each rule r_i in *in_R*, yielding a function *kappaMinus* mapping rules to natural numbers. For each world w, *sumPenalty(w, in_R, kappaMinus)* computes the sum of the κ_i^- values according to equation 3, and, if necessary, *kappaNull* is determined as a negative normalization constant so that at least one world gets rank 0 in the resulting state *kappa*.

5 Conclusions and Further Work

Condor@AsmL is a fully operational system modelling an intelligent agent being capable to reason about default rules and to adapt its own state of belief when new information arrives. AsmL provided a very helpful tool for the implementation of Condor@AsmL by minimizing the gap between abstract specification and executable code, which was the main objective of its design. Efficiency considerations and the use of e.g. knowledge base compilation techniques are topics of further work.

References

1. Alchourrón, C.E., Gärdenfors, P., Makinson, P.: On the logic of theory change: Partial meet contraction and revision functions. Journal of Symbolic Logic 50(2), 510–530 (1985)
2. AsmL webpage (2007), http://research.microsoft.com/foundations/asml/
3. Beierle, C., Kern-Isberner, G.: An ASM refinement and implementation of the Condor system using ordinal conditional functions. In: Prinz, A. (ed.) Proceedings 14th International Workshop on Abstract State Machines (ASM 2007), Agder University College, Grimstad, Norway (2007)
4. Benferhat, S., Dubois, D., Prade, H.: Representing default rules in possibilistic logic. In: Proceedings 3th International Conference on Principles of Knowledge Representation and Reasoning KR 1992, pp. 673–684 (1992)
5. DeFinetti, B.: Theory of Probability, vol. 1, 2. John Wiley & Sons, Chichester (1974)
6. Goldszmidt, M., Pearl, J.: Qualitative probabilities for default reasoning, belief revision, and causal modeling. Artificial Intelligence 84, 57–112 (1996)
7. Gurevich, Y., Rossman, B., Schulte, W.: Semantic essence of AsmL. Theoretical Computer Science 343(3), 370–412 (2005)
8. Kern-Isberner, G.: A thorough axiomatization of a principle of conditional preservation in belief revision. Annals of Mathematics and Artificial Intelligence 40(1-2), 127–164 (2004)
9. Spohn, W.: Ordinal conditional functions: a dynamic theory of epistemic states. In: Harper, W.L., Skyrms, B. (eds.) Causation in Decision, Belief Change, and Statistics, II, pp. 105–134. Kluwer Academic Publishers, Dordrecht (1988)

The Abella Interactive Theorem Prover (System Description)

Andrew Gacek

Department of Computer Science and Engineering, University of Minnesota
200 Union Street SE, Minneapolis, MN 55455, USA

1 Introduction

Abella [3] is an interactive system for reasoning about aspects of object languages that have been formally presented through recursive rules based on syntactic structure. Abella utilizes a two-level logic approach to specification and reasoning. One level is defined by a specification logic which supports a transparent encoding of structural semantics rules and also enables their execution. The second level, called the reasoning logic, embeds the specification logic and allows the development of proofs of properties about specifications. An important characteristic of both logics is that they exploit the λ-tree syntax approach to treating binding in object languages. Amongst other things, Abella has been used to prove normalizability properties of the λ-calculus, cut admissibility for a sequent calculus and type uniqueness and subject reduction properties. This paper discusses the logical foundations of Abella, outlines the style of theorem proving that it supports and finally describes some of its recent applications.

2 The Logic Underlying Abella

Abella is based on $\mathcal{G}$, an intuitionistic, predicative, higher-order logic with fixed-point definitions for atomic predicates and with natural number induction [4].

Representing binding. $\mathcal{G}$ uses the λ*-tree syntax* approach to representing syntactic structures [7], which allows object level binding to be represented using meta-level abstraction. Thus common notions related to binding such as α-equivalence and capture-avoiding substitution are built into the logic, and the encodings of object languages do not need to implement such features.

To reason over λ-tree syntax, $\mathcal{G}$ uses the ∇ quantifier which represents a notion of generic judgment [9]. A formula $\nabla x.F$ is true if F is true for each x in a generic way, *i.e.*, when nothing is assumed about any x. This is a stronger statement than $\forall x.F$ which says that F is true for all values for x but allows this to be shown in different ways for different values.

For the logic $\mathcal{G}$, we assume the following two properties of ∇:

$$\nabla x.\nabla y.F\ x\ y \equiv \nabla y.\nabla x.F\ x\ y \qquad \nabla x.F \equiv F \text{ if } x \text{ not free in } F$$

A natural proof-theoretic treatment for this quantifier is to use *nominal constants* to instantiate ∇-bound variables [16]. Specifically, the proof rules for ∇ are

$$\frac{\Gamma, B[a/x] \vdash C}{\Gamma, \nabla x.B \vdash C}\ \nabla\mathcal{L} \qquad\qquad \frac{\Gamma \vdash C[a/x]}{\Gamma \vdash \nabla x.C}\ \nabla\mathcal{R}$$

where a is a nominal constant which does not appear in the formula underneath the ∇ quantifier. Due to the equivalence of permuting ∇ quantifiers, nominal constants must be treated as permutable, which is captured by the initial rule.

$$\frac{\pi.B = B'}{\Gamma, B \vdash B'}\ id_\pi$$

Here π is a permutation of nominal constants.

Definitions. The logic $\mathcal{G}$ supports fixed-point definitions of atomic predicates. These definitions are specified as clauses of the form $\forall\overline{x}.(\nabla\overline{z}.H) \triangleq B$ where the head H is an atomic predicate. This notion of definition is extended from previous notions (*e.g.*, see [9]) by admitting the ∇-quantifier in the head. Roughly, when such a definition is used, in ways to be explained soon, these ∇-quantified variables become instantiated with nominal constants from the term on which the definition is used. The instantiations for the universal variables $\overline{x}$ may contain any nominal constants not assigned to the variables $\overline{z}$. Thus ∇ quantification in the head of a definition allows us to restrict certain pieces of syntax to be nominal constants and to state dependency information for those nominal constants.

Two examples hint at the expressiveness of our extended form of definitions. First, we can define a predicate *name* E which holds only when E is a nominal constant. Second, we can define a predicate *fresh* X E which holds only when X is a nominal constant which does not occur in E.

$$(\nabla x.\mathit{name}\ x) \triangleq \top \qquad\qquad \forall E.(\nabla x.\mathit{fresh}\ x\ E) \triangleq \top$$

Note that the order of quantification in *fresh* enforces the freshness condition.

Definitions can be used in both a positive and negative fashion. Positively, definitions are used to derive an atomic judgment, *i.e.*, to show a predicate holds on particular values. This use corresponds to unfolding a definition and is similar to back-chaining. Negatively, an atomic judgment can be decomposed in a case analysis-like way based on a closed-world reading of definitions. In this case, the atomic judgment is unified with the head of each definitional clause, where eigenvariables are treated as instantiatable. Also, both the positive and negative uses of definitions consider permutations of nominal constants in order to allow the ∇-bound variables $\overline{z}$ to range over any nominal constants. A precise presentation of these rules, which is provided in Gacek *et al.* [4], essentially amounts to introduction rules for atomic judgments on the right and left sides of sequents in a sequent calculus based presentation of the logic.

Induction. $\mathcal{G}$ supports induction over natural numbers. By augmenting the predicates being defined with a natural number argument, this induction can serve as a method of proof based on the length of a bottom-up evaluation of a definition.

$$\forall m, n, a, b[\mathit{of}\, m\ (\mathit{arr}\ a\ b) \wedge \mathit{of}\, n\ a\ \supset\ \mathit{of}\, (\mathit{app}\ m\ n)\ b]$$
$$\forall r, a, b[\forall x[\mathit{of}\, x\ a \supset \mathit{of}\, (r\ x)\ b] \supset \mathit{of}\, (\mathit{abs}\ a\ r)\ (\mathit{arr}\ a\ b)]$$

Fig. 1. Second-order hereditary Harrop formulas for typing

3 The Structure of Abella

The architecture of Abella has two distinguishing characteristics. First, Abella is oriented towards the use of a specific (executable) specification logic whose proof-theoretic structure is encoded via definitions in $\mathcal{G}$. Second, Abella provides tactics for proof construction that embody special knowledge of the specification logic. We discuss these aspects and their impact in more detail below.

3.1 Specification Logic

It is possible to encode object language descriptions directly in definitions in $\mathcal{G}$, but there are two disadvantages to doing so: the resulting definitions may not be executable and there are common patterns in specifications with λ-tree syntax which we would like to take advantage of. We address these issues by selecting a specification logic which has the features that the $\mathcal{G}$ lacks, and embedding the evaluation rules of this specification logic instead into $\mathcal{G}$. Object languages are then encoded through descriptions in the specification logic [6].

The specification logic of Abella is second-order hereditary Harrop formulas [8] with support for λ-tree syntax. This allows a transparent encoding of structural operational semantics rules which operate on objects with binding. For example, consider the simply-typed λ-calculus where types are either a base type i or arrow types constructed with *arr*. Terms are encoded with the constructors *app* and *abs*. The constructor *abs* takes two arguments: the type of the variable being abstracted and the body of the function. Rather than having a constructor for variables, the body argument to *abs* is an abstraction in our specification logic, thus object level binding is represented by the specification logic binding. For example, the term $(\lambda f\!:\!i \to i.(\lambda x\!:\!i.(f\ x)))$ is encoded as

$$\mathit{abs}\ (\mathit{arr}\ i\ i)\ (\lambda f.\mathit{abs}\ i\ (\lambda x.\mathit{app}\ f\ x)).$$

In the latter term, λ denotes an abstraction in the specification logic. Given this representation, the typing judgment $\mathit{of}\, m\ t$ is defined in Figure 1. Note that these rules do not maintain an explicit context for typing assumptions, instead using a hypothetical judgment to represent assumptions. Also, there is no side-condition in the rule for typing abstractions to ensure the variable x does not yet occur in the typing context, since instead of using a particular x for recording a typing assumption, we quantify over all x.

Our specification of typing assignment is executable. More generally, the Abella specification logic is a subset of the language λProlog [11] which can be compiled and executed efficiently [12]. This enables the animation of specifications, which is convenient for assessing specifications before attempting to prove properties over them. This also allows specifications to be used as testing oracles when developing full implementations.

The evaluation rules of our specification logic can be encoded as a definition in $\mathcal{G}$. A particular specification is then encoded in a separate definition which is used by the definition of evaluation in order to realize back-chaining over specification clauses. Reasoning over a specification is realized by reasoning over its evaluation via the definition of the specification logic. Abella takes this further and is customized towards the specification logic. For example, the context of hypothetical judgments in our specification logic admits weakening, contraction, and permutation, all of which are provable in $\mathcal{G}$. Abella automatically uses this meta-level property of the specification logic during reasoning. Details on the benefits of this approach to reasoning are available in Gacek *et al.* [5].

3.2 Tactics

The user constructs proofs in Abella by applying tactics which correspond to high-level reasoning steps. The collection of tactics can be grouped into those that generically orchestrate the rules of $\mathcal{G}$ and those that correspond to meta-properties of the specification logic. We discuss these classes in more detail below.

Generic tactics. The majority of tactics in Abella correspond directly to inference rules in $\mathcal{G}$. The most common tactics from this group are the ones which perform induction, introduce variables and hypotheses, conduct case analysis, apply lemmas, and build results from hypotheses. In the examples suite distributed with Abella, these five tactics make up more than 90% of all tactic usages. The remaining generic tactics are for tasks such as splitting a goal of the form $G_1 \wedge G_2$ into two separate goals for G_1 and G_2, or for instantiating the quantifier in a goal of the form $\exists x.G$.

Specification logic tactics. Since our specification logic is encoded in $\mathcal{G}$, we can formally prove meta-level properties for it. Once such properties are proved, their use in proofs can be built into tactics. Two important properties that Abella uses in this way are instantiation and cut admissibility. In detail, negative uses of the specification logic $\forall$ quantifier are represented in $\mathcal{G}$ as nominal constants (*i.e.*, the ∇ quantifier), and the instantiation tactic allows such nominal constants to be instantiated with specific terms. The cut tactic allows hypothetical judgments to be relieved by showing that they are themselves provable.

4 Implementation

Abella is implemented in OCaml. The most sophisticated component of this implementation is higher-order unification which is a fundamental part of the logic $\mathcal{G}$. It underlies how case analysis is performed, and in the implementation, unification is used to decide when tactics apply and to determine their result. Thus an efficient implementation of higher-order unification is central to an efficient prover. For this, Abella uses the the higher-order pattern unification package of Nadathur and Linnell [10]. We have also extended this package to deal with the particular features and consequences of reasoning in $\mathcal{G}$.

Treatment of nominal constants. As their name suggests, nominal constants can be treated very similarly to constants for most of the unification algorithm, but there are two key differences. First, while traditional constants can appear in the instantiation of variables, nominal constants cannot appear in the instantiation of variables. Thus dependency information on nominal constants is tracked via explicit raising of variables. Second, nominal constants can be permuted when determining unifiability. However, even in our most sophisticated examples the number of nominal constants appearing at the same time has been at most two. Thus, naive approaches to handling permutability of nominal constants have sufficed and there has been little need to develop sophisticated algorithms.

Simple extensions. The treatment of case analysis via unification for eigenvariables creates unification problems which fall outside of the higher-order pattern unification fragment, yet still have most general unifiers. For example, consider the clause for β-contraction in the λ-calculus:

$$\mathit{step}\ (\mathit{app}\ (\mathit{abs}\ R)\ M)\ (R\ M).$$

Case analysis on a hypotheses of the form $\mathit{step}\ A\ B$ will result in the attempt to solve the unification problem $B = R\ M$ where B, R, and M are all instantiatable. This is outside of the higher-order pattern unification fragment since R is applied to an instantiatable variable, but there is a clear most general unifier. When nominal constants are present, this situation is slightly more complicated with unification problems such as $B\ x = R\ M\ x$ or $B\ x = R\ (M\ x)$, where x is a nominal constant. The result is the same, however, that a most general unifier exists and is easy to find.

5 Examples

This section briefly describes sample reasoning tasks we have conducted in Abella. The detailed proofs are available in the distribution of Abella [3].

Results from the λ-calculus. Over untyped λ-terms, we have shown the equivalence of big-step and small-step evaluation, preservation of typing for both forms of evaluation, and determinacy for both forms of evaluation. We have shown that the λ-terms can be disjointly partitioned into normal and non-normal forms. Over simply-typed λ-terms, we have shown that typing assignments are unique.

Cut admissibility. We have shown that the cut rule is admissible for a sequent calculus with implication and conjunction. The representation of sequents in our specification logic used hypothetical judgments to represent hypotheses in the sequent. This allowed the cut admissibility proof to take advantage of Abella's built-in treatment of meta-properties of the specification logic.

The POPLmark challenge. The POPLmark challenge [1] is a selection of problems which highlight the traditional difficulties in reasoning over systems which manipulate objects with binding. The particular tasks of the challenge involve reasoning about evaluation, typing, and subtyping for $F_{<:}$, a λ-calculus with bounded subtype polymorphism. We have solved parts 1a and 2a of this challenge using Abella, which represent the fundamental reasoning tasks involving objects with binding.

Proving normalizability à la Tait. We have shown that all closed terms in the call-by-value, simply-typed λ-calculus are normalizable using the logical relations argument in the style of Tait [14]. Fundamental in this proof was the encoding of arbitrary cascading substitutions which allows one to consider all closed instantiations for an open λ-term. Encoding and reasoning over this form of substitution makes essential use of the extended form of definitions in $\mathcal{G}$.

6 Future and Related Work

Induction and coinduction The logic $\mathcal{G}$ currently supports induction on natural numbers. Similar logics have been extended to support structural induction and coinduction on definitions [15]. Already, the implementation of Abella has support for these features. A paper which describes the extended logic supporting these features is in preparation.

User programmability. Tactics-based theorem provers often support *tacticals* which allow users to compose tactics in useful ways. Some systems even go beyond this and offer a full programming language for creating custom tactics. We would like to extend Abella with such features.

Proof search. Many proofs in Abella follow a straightforward pattern of essentially induction, case analysis, and building from hypotheses. We would like to extend Abella to perform these types of proofs automatically. Recent results on focusing in similar logics may offer some insight into a disciplined approach to automated proof search [2].

Related work. A closely related system is Twelf [13] which is based on a dependently typed λ-calculus for specification. Controlling for dependent types, the most significant difference is that our meta-logic is significantly richer than the one in Twelf. Also related is the Nominal package [17] for Isabelle/HOL which allows for reasoning over α-equivalence classes. This approach leverages on existing theorem proving work, but does not address the full problem of reasoning with binding. In particular, all work related to substitution is left to the user. A more detailed comparison with these works is available in Gacek *et al.* [5].

Acknowledgements

I am grateful to David Baelde, Dale Miller, Gopalan Nadathur, Randy Pollack, and Alwen Tiu for their input and feedback on the development of Abella.

Anonymous reviewers provided helpful comments on an earlier version of this paper. This work has been supported by the NSF Grant CCR-0429572 and by a grant from Boston Scientific. Opinions, findings, and conclusions or recommendations expressed in this work are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

1. Aydemir, B.E., Bohannon, A., Fairbairn, M., Foster, J.N., Pierce, B.C., Sewell, P., Vytiniotis, D., Washburn, G., Weirich, S., Zdancewic, S.: Mechanized metatheory for the masses: The PoplMark challenge. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 50–65. Springer, Heidelberg (2005)
2. Baelde, D., Miller, D.: Least and greatest fixed points in linear logic. In: Dershowitz, N., Voronkov, A. (eds.) LPAR 2007. LNCS (LNAI), vol. 4790, pp. 92–106. Springer, Heidelberg (2007)
3. Gacek, A.: The Abella system. Available in source code (2008), http://abella.cs.umn.edu/
4. Gacek, A., Miller, D., Nadathur, G.: Combining generic judgments with recursive definitions. In: Pfenning, F. (ed.) 23th Symp. on Logic in Computer Science. IEEE Computer Society Press, Los Alamitos (to appear, 2008)
5. Gacek, A., Miller, D., Nadathur, G.: Reasoning in Abella about structural operational semantics specifications. In: LFMTP 2008 (to appear, 2008), http://arxiv.org/abs/0804.3914
6. McDowell, R., Miller, D.: Reasoning with higher-order abstract syntax in a logical framework. ACM Trans. on Computational Logic 3(1), 80–136 (2002)
7. Miller, D.: Abstract syntax for variable binders: An overview. 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. LNCS (LNAI), vol. 1861, pp. 239–253. Springer, Heidelberg (2000)
8. Miller, D., Nadathur, G., Pfenning, F., Scedrov, A.: Uniform proofs as a foundation for logic programming. Annals of Pure and Applied Logic 51, 125–157 (1991)
9. Miller, D., Tiu, A.: A proof theory for generic judgments. ACM Trans. on Computational Logic 6(4), 749–783 (2005)
10. Nadathur, G., Linnell, N.: Practical higher-order pattern unification with on-the-fly raising. In: Gabbrielli, M., Gupta, G. (eds.) ICLP 2005. LNCS, vol. 3668, pp. 371–386. Springer, Heidelberg (2005)
11. Nadathur, G., Miller, D.: An Overview of λProlog. In: Fifth International Logic Programming Conference, Seattle, August 1988, pp. 810–827. MIT Press, Cambridge (1988)
12. Nadathur, G., Mitchell, D.J.: System description: Teyjus—A compiler and abstract machine based implementation of Lambda Prolog. In: Ganzinger, H. (ed.) Proceedings of the 16th International Conference on Automated Deduction, Trento, Italy, July 1999, pp. 287–291. Springer, Heidelberg (1999)
13. Pfenning, F., Schürmann, C.: System description: Twelf — A meta-logical framework for deductive systems. In: Ganzinger, H. (ed.) 16th Conference on Automated Deduction. LNCS (LNAI), pp. 202–206. Springer, Heidelberg (1999)
14. Tait, W.W.: Intensional interpretations of functionals of finite type I. J. of Symbolic Logic 32(2), 198–212 (1967)

15. Tiu, A.: A Logical Framework for Reasoning about Logical Specifications. PhD thesis, Pennsylvania State University (May 2004)
16. Tiu, A.: A logic for reasoning about generic judgments. In: Momigliano, A., Pientka, B. (eds.) International Workshop on Logical Frameworks and Meta-Languages:Theory and Practice (LFMTP 2006) (2006)
17. Urban, C., Tasson, C.: Nominal techniques in Isabelle/HOL. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 38–53. Springer, Heidelberg (2005)

LEO-II - A Cooperative Automatic Theorem Prover for Classical Higher-Order Logic (System Description)

Christoph Benzmüller[1], Lawrence C. Paulson[2], Frank Theiss[1], and Arnaud Fietzke[3]

[1]Dep. of Computer Science, Saarland University, Saarbrücken, Germany
[2]Computer Laboratory, The University of Cambridge, UK
[3]Max Planck Institute for Informatics, Saarbrücken, Germany

Abstract. LEO-II is a standalone, resolution-based higher-order theorem prover designed for effective cooperation with specialist provers for natural fragments of higher-order logic. At present LEO-II can cooperate with the first-order automated theorem provers E, SPASS, and Vampire. The improved performance of LEO-II, especially in comparison to its predecessor LEO, is due to several novel features including the exploitation of term sharing and term indexing techniques, support for primitive equality reasoning, and improved heuristics at the calculus level. LEO-II is implemented in Objective Caml and its problem representation language is the new TPTP THF language.

1 Introduction and Motivation

Automatic theorem provers (ATPs) based on the resolution principle, such as Vampire [20], E [21], and SPASS [25], have reached a high degree of sophistication. They can often find long proofs even for problems having thousands of axioms. However, they are limited to first-order (FO) logic. Higher-order (HO) logic extends FO logic with lambda notation for functions and with function and predicate variables. It supports reasoning in set theory, using the obvious representation of sets by predicates. HO logic is a natural language for expressing mathematics, and it is also ideal for formal verification. Moving from FO to HO logic requires a more complicated proof calculus, but it often allows much simpler problem statements. HO logic's built-in support for functions and sets often leads to shorter proofs. Conversely, elementary identities (such as the distributive law for union and intersection) turn into difficult problems when expressed in FO form.

LEO-II is a standalone, resolution-based HO ATP that is designed for fruitful cooperation with specialist provers for fragments of HO logic. The idea is to combine the strengths of the different systems. On the other hand, LEO-II itself, as an external reasoner, is designed to support HO proof assistants such as Isabelle/HOL [18], HOL [13], and ΩMEGA [22] by efficiently automating subproblems and thereby reducing user effort.

1.1 Motivation for LEO-II

LEO-II is the successor of LEO [4], which was implemented in Allegro Common Lisp as a part of the ΩMEGA system and which unfortunately was not available as a standalone reasoner. LEO shared the basic data structures such as terms and clauses with ΩMEGA; these shared basic data structures were not designed for efficiency and their term indexing support was limited [15]. LEO's performance strongly improved after coupling it with the FO ATP Otter in the agent based ΩANTS framework [8]. This integration has subsequently been improved and Otter has been replaced by Bliksem and Vampire [9]. This cooperative theorem proving approach outperforms – modulo different problem representations – FO ATPs such as Vampire on problems about sets, relations, and functions as given in the TPTP SET domain (see Figure 1). The cooperative approach not only solves more problems in this domain; it also solves them more efficiently. This provides evidence for the following working hypothesis:

It is well known in AI that representation matters. Problem representation particularly matters in automated theorem proving: many proof problems can be effectively solved when they are initially represented in a natural way in HO logic and then tackled with a cooperative theorem proving approach in which a HO reasoner subsequently reduces them to a suitable fragment of HO logic in which they can be tackled by an effective specialist reasoner.

So far, we consider only FO logic as a target fragment. The general idea, however, is not limited to this; other fragments will be studied in future work. Examples include decidable fragments like monadic second-order logic and the guarded fragment. Decidability can probably produce useful feedback for improved heuristic control in the HO ATP.

2 Overview of LEO-II

The one year project "LEO-II: An Effective Higher-Order Theorem Prover" was funded by EPSRC at Cambridge University, UK under grant EP/D070511/1. In this project LEO-II has been (re-)implemented in Objective Caml 3.09 (approx. 12500 lines of code) as a standalone HO ATP. LEO-II can be easily installed, deployed and integrated with other reasoners and its sources are available from the LEO-II web-site at: `http://www.ags.uni-sb.de/~leo/`.

Proof search in LEO-II, which is based on the extensional HO resolution calculus [3] has been further improved, e.g., it now supports efficient (recursive) expansion of definitions and primitive equality reasoning (see Section 3). For cooperation with FO ATPs, LEO-II offers two different HO to FO mappings; further mappings can easily be added.

LEO-II provides efficient term data structures. It employs perfect term sharing and supports for HO term indexing. LEO-II also provides analysis tools for exploring its proof object, term graph and term index. This includes statistical analysis of the term graph (see Section 4).

In addition to a fully automatic mode, LEO-II also provides an interactive mode [6]. This mode supports debugging and inspection of the search space,

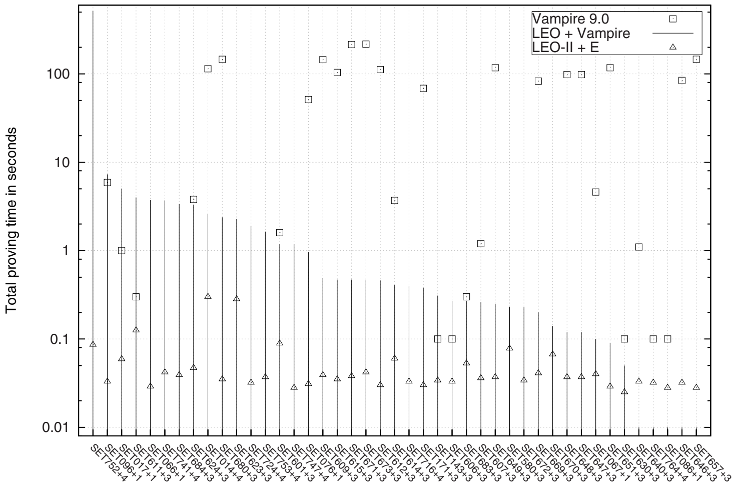


Fig. 1. The new LEO-II+E cooperation strongly outperforms the old LEO+Vampire cooperation on HO encodings of proof problems as given in the TPTP SET domain (these HO encodings are distributed with LEO-II). The worst approach is to tackle these problems in their original, less elegant FO encoding with Vampire.

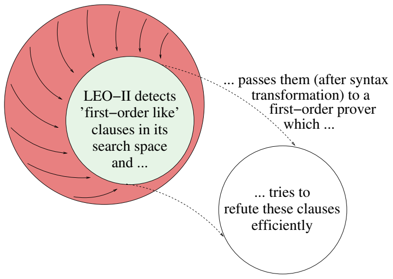
but also the tutoring of resolution based HO theorem proving to students. The interactive mode and the automatic mode can be interleaved.

LEO-II supports prefix-polymorphism; full polymorphism adds many non-trivial choice points to the already challenging search space of LEO-II [6] and is therefore future work. Moreover, LEO-II employs the new TPTP THF representation language [7].

At present, LEO-II cooperates with FO ATPs only in a sequential mode and not via the agent based architecture *Ω*Ants. Figure 1 shows that LEO-II's performance has nevertheless strongly improved (in our experiment version 0.95 of LEO-II was cooperating with E 0.99 "Singtom").

In the remainder we sketch LEO-II's cooperative proof search (Section 3) and its term sharing and term indexing (Section 4).

3 Cooperative Proof Search



LEO-II's clause set generally consists of HO clauses (dark-colored area), which are processed with LEO-II's calculus rules. Some of the clauses in LEO-II's search space additionally attain a special status (the ones in the light-colored area): they are in the domain of a transformation function to a particular fragment of HO logic, here FO logic. Light-colored clauses are available to both LEO-II's proof search and to a specialist prover for the target fragment via the transformation function. Roughly speaking, currently all clauses that do not contain any λ-term and embedded predicate or proposition are light-colored, and our default FO transformation function employs Kerber's ideas [14]: it recursively translates every application $(p_{\alpha\rightarrow\beta}\, q_\alpha)$ into a term $@^{(\alpha\rightarrow\beta)\text{-}\alpha}(p,q)$, where $@^{(\alpha\rightarrow\beta)\text{-}\alpha}$ is a new function or predicate constant that encodes the types of its two arguments. LEO-II's extensional HO resolution approach, which enhances standard resolution proof search with specific extensionality rules, is explained in detail in [3] and [6]. It is well suited to subsequently generate more and more light-colored clauses from dark colored ones. In some proof problems (such as Examples 1 and 2 below) the light-colored area quickly collects enough information for constructing a refutation; however, LEO-II is often too weak to find this refutation on its own. In other proof problems (see Example 3 below) the refutation can be found only in the dark-colored area. This observation motivates a distributed architecture in which LEO-II dynamically cooperates with incremental specialist reasoners. At present, LEO-II sequentially launches a fresh call of the cooperating specialist prover every n iterations of its (standard) resolution proof search loop (currently $n = 10$).

Next, we discuss three proof problems from the domain of normal multimodal logics. For this domain, our cooperative approach yields elegant problem encodings and efficient solutions [5].

The encoding of multimodal logic in simple type theory is intuitive. Choose a base type — we choose ι — to denote the set of all possible worlds. Certain formulas of type $\iota \rightarrow o$ then correspond to multimodal logic expressions. The modal operators $\neg$, $\vee$, and $\Box_R$ become λ-terms of types $(\iota \rightarrow o) \rightarrow (\iota \rightarrow o)$, $(\iota \rightarrow o) \rightarrow (\iota \rightarrow o) \rightarrow (\iota \rightarrow o)$, and $(\iota \rightarrow \iota \rightarrow o) \rightarrow (\iota \rightarrow o) \rightarrow (\iota \rightarrow o)$ respectively. Note that $\neg$ forms the complement of a set of worlds, while $\vee$ forms the union of two such sets. $\Box_R$ explicitly abstracts over the accessibility relation R:

$$\begin{aligned}
\neg_{(\iota\rightarrow o)\rightarrow(\iota\rightarrow o)} &= \lambda A_{\iota\rightarrow o} \centerdot \lambda X_\iota \centerdot \neg A\, X \\
\vee_{(\iota\rightarrow o)\rightarrow(\iota\rightarrow o)\rightarrow(\iota\rightarrow o)} &= \lambda A_{\iota\rightarrow o} \centerdot \lambda B_{\iota\rightarrow o} \centerdot \lambda X_\iota \centerdot A\, X \vee B\, X \\
\Box_{R\ (\iota\rightarrow\iota\rightarrow o)\rightarrow(\iota\rightarrow o)\rightarrow(\iota\rightarrow o)} &= \lambda R_{\iota\rightarrow\iota\rightarrow o} \centerdot \lambda A_{\iota\rightarrow o} \centerdot \lambda X_\iota \centerdot \forall X_\iota \centerdot R\, X\, Y \Rightarrow A\, X
\end{aligned}$$

A multimodal logic formula $A_{\iota\rightarrow o}$ is valid iff for all possible worlds W_ι we have $W \in A$, that is, iff $A\,W$ holds: **valid** $= \lambda A_{\iota\rightarrow o} \centerdot \forall W_\iota \centerdot A\, W$. Reflexivity and transitivity are defined as (we omit types) **refl** $= \lambda R \centerdot \forall X \centerdot R\, X\, X$ and **trans** $= \lambda R \centerdot \forall X \centerdot \forall Y \centerdot \forall Z \centerdot R\, X\, Y \wedge R\, Y\, Z \Rightarrow R\, X\, Z$. More details on this encoding, which models multimodal logic **K**, can be found in [5].

Example 1 (A simple equation between modal logic formulas).
$\forall R \centerdot \forall A \centerdot \forall B \centerdot (\Box_R\,(A \vee B)) = (\Box_R\,(B \vee A))$

Example 2. [The well known multimodal axioms $\Box_R\, A \Rightarrow A$ and $\Box_R\, A \Rightarrow \Box_R\, \Box_R\, A$ are equivalent to reflexivity and transitivity of the accessibility relation R]
$\forall R \centerdot (\forall A \centerdot \mathbf{valid}(\Box_R\, A \Rightarrow A) \wedge \mathbf{valid}(\Box_R\, A \Rightarrow \Box_R\, \Box_R\, A)) \Leftrightarrow (\mathbf{refl}(R) \wedge \mathbf{trans}(R))$

Example 3 (Axiom T is not valid).
$\neg\forall R_{\blacksquare}\forall A_{\blacksquare}(\mathbf{valid}(\Box_R A \Rightarrow A))$

After negating the statement in Example 1, recursive expansion of the definitions, and exhaustive clause normalization (with integrated functional and Boolean extensionality treatment), LEO-II generates ten light-colored clauses. Amongst them are $(b\,V) \vee (a\,V) \vee \neg((r\,w)\,V) \vee \neg((r\,w)\,U) \vee (b\,U) \vee (a\,U)$ and $\neg(a\,z) \vee \neg(b\,v)$ and $((r\,w)\,z) \vee ((r\,w)\,v)$ where upper case and lower case symbols denote variables and Skolem constants, respectively.[1] This light-colored clause set is immediately refutable by E, so that LEO-II does not even start its main proof loop. (The total proving time is $0.071s$ on a Linux notebook with 1.60GHz, 1GB memory.) Definition expansion and normalization does not always produce refutable light-colored clauses: When replacing the primitive equality $=$ in Example 1 by Leibniz equality $\lambda X, Y.\forall P_{\blacksquare}(P\,X) \Rightarrow (P\,Y)$, we obtain the dark-colored clauses $(p\,(\lambda X_\iota.\forall Y_{\iota\blacksquare}\neg((r\,X)\,Y) \vee (a\,Y) \vee (b\,Y)))$ and $\neg(p\,(\lambda X_\iota.\forall Y_{\iota\blacksquare}\neg((r\,X)\,Y) \vee (b\,Y) \vee (a\,Y)))$. LEO-II starts its proof search and applies resolution and extensional unification [3] to them which subsequently returns a refutable set of light-colored clauses as above. (The total proving time is $0.166s$.)

Example 2 is more challenging than Example 1. Proof search in LEO-II, however, is analogous with one crucial difference: definition expansion and normalization generates 70 clauses and E generates more than 20 000 clauses from them before finding the refutation. (The total proving time is $2.48s$.) This example illustrates the benefit of the cooperation, since LEO-II alone is not able to find this refutation in its search space: LEO-II generates too many clauses and gets stuck; moreover, LEO-II is still incomplete even for the FO fragment.

In Example 3, the refutation is found only in the dark-colored area. (The total proving time is $9.02s$.) Definition expansion and normalization generates the clauses $((R\,W)\,s^{A,W,R}) \vee (A\,W)$ and $\neg(A\,s^{A,W,R}) \vee (A\,W)$, where $s^{A,W,R} = (((s\,A)\,W)\,R)$ is a new Skolem term. The refutation employs only the former clause. LEO-II applies its primitive substitution rule [3,6] to guess the instantiations $R \leftarrow \lambda X, Y.((M\,X)\,Y) \neq ((N\,X)\,Y)$ and $A \leftarrow \lambda X.(OX) \neq (PX)$ where M, N, O, P are new free variables. (Hence, LEO-II proposes to consider inequality for the the accessibility relation, which is not reflexive and does not satisfy axiom T.) Applying this instantiation leads to two unification constraints, i.e., negated equation literals in LEO-II, which both have flexible term heads. Proof search terminates since such flex-flex unification constraints are always solvable.

It is not obvious how the examples in this section can be represented in first-order logic. Therefore, these examples (and many others we are currently studying) are not included in the comparison in Figure 1.

LEO-II shows some promising first results on examples as presented in this paper, though it is not yet complete for simple type theory. Two known sources

[1] To illustrate the use of FO TPTP we show how the latter clause is represented modulo our transformation function in FO TPTP syntax (with T encoding brackets):
$fof(leo\_II\_clause\_177, axiom,$
$(at\_io\_i(at\_iTioT\_i(sk1, sk4), sk13)|at\_io\_i(at\_iTioT\_i(sk1, sk4), sk11)))$.

of incompleteness are LEO-II's pruning of unification problems at a preset unification depth (nested projections and imitations) and its limited support for factorization modulo unification. In simple examples with Church numerals the former restriction, for example, prevents LEO-II from synthesizing Church numerals beyond this depth and because of the latter restriction LEO-II still cannot solve the famous Cantor theorem. Future work will address these aspects.

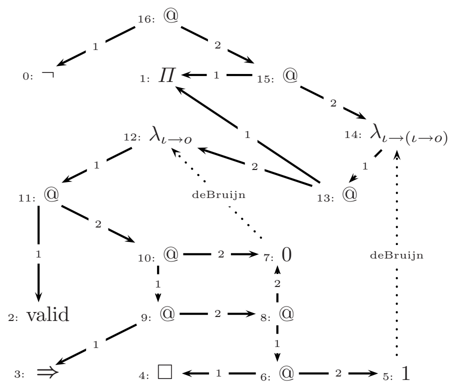
4 Term Sharing and Term Indexing

Term indexing techniques are widely used in major FO ATPs [20,21,25]. The indexing data structures store large numbers of terms and, for a given query term t, support the fast retrieval of terms from the index that satisfy a certain relation with t, such as matching, unifiability or syntactic equality [17]. Performance can be further enhanced by representation of terms in efficient data structures, such as shared terms, used for instance in E [21].

HO term indexing techniques are rarely addressed in the literature, which hampers the progress of systems in this field. An exception is Pientka [19].

LEO-II's implementation at term level is based on a perfectly shared term graph, i.e., syntactically equal terms are represented by a single instance. Ideas from FO term sharing are adapted to HO logic by (i) keeping indexed terms in $\beta\eta$ normal form (i.e., η short and β normal) and (ii) using de Bruijn indices [12] to allow λ-abstracted terms to be shared.

The resulting data structure represents terms in a directed acyclic graph (DAG). LEO-II supports the visualization of such term graphs. The graph to the right shows the (not heavily shared) term graph after initialization of Example 3.[2]



LEO-II also supports statistical analysis of its term graphs (Figure 2). Future work will investigate whether such information can be exploited for improving heuristic control.

Representation of terms in a shared graph naturally advances the performance of a number of operations, e.g., it allows fast lookup of all occurrences of syntactically equal terms or subterms, and it improves the performance of rewrite operations, such as global unfolding of definitions. Additionally, LEO-II employs a term indexing data structure, which is

[2] To further visualize the evolution of the term graph during proof search, we modified LEO-II to output a snapshot of its state after each processing step. This data was used to create animations of dynamically changing term graphs during proof search. The video clips can be obtained at `http://www.ags.uni-sb.de/~leo/art.html`

```
------------ The Termset Analysis -------------
[...]

Sharing rate: 17 nodes with 18 bindings
Average sharing rate (bindings per node):          1.05
Average term size:                                 6.58
Average number of supernodes:                      5.47
Average number of supernodes (symbols):            5.80
Average number of supernodes (nonprimitive terms): 4.50
Rate of term occurrences PST size / term size:     0.36
Rate of symbol occurrences PST size / term size:   0.39
Rate of bound occurrences PST size / term size:    0.57
------------ End Termset Analysis ------------
```

```
------------ The Termset Analysis -------------
[...]

Sharing rate: 2094 nodes with 3415 bindings
Average sharing rate (bindings per node):           1.63
Average term size:                                 13.95
Average number of supernodes:                       9.47
Average number of supernodes (symbols):            28.83
Average number of supernodes (nonprimitive terms):  5.73
Rate of term occurrences PST size / term size:      0.24
Rate of symbol occurrences PST size / term size:    0.30
Rate of bound occurrences PST size / term size:     0.52
------------ End Termset Analysis ------------
```

Fig. 2. Statistical analysis of term sharing aspects in LEO-II after initialisation of Example 3 (left) and after the proof has been found (right)

based on structural indexing methods from the FO domain [16,23], as well as road sign techniques. Road signs are features of the data structure which guide operations based on graph traversal. They help to cut branches of the subgraph to be processed early and they are employed, e.g., in the construction of partial syntax trees [24] in which all branches with no occurrences of a given symbol or subterm are cut. This enables LEO-II to avoid potentially costly operations, such as occurs checks, and to speed up basic operations on terms, such as substitution.

Future work includes the development, comparison and evaluation of other termindexing techniques within LEO-II.

5 Conclusion

LEO-II, which replaces the previous LEO system, realizes a cooperative HO-FO theorem proving approach that shows some promising results in selected problem domains. It thus provides an interesting alternative to reasoning solely in FO logic and it also differs from other HO ATPs such as TPS [1] and OTTER-λ [2]. TPS is based on the mating method and does not cooperate with specialist FO ATPs. It is particularly strong in proving theorems which require selective expansion of definitions and goal directed instantiation of set variables. Examples of such theorems are presented in Bishop and Andrews [10] and Brown [11]. Many of these examples will require corresponding mechanisms in LEO-II to be proven automatically. The formalism of OTTER-λ is not simple type theory but λ-logic, which is a novel combination of λ-calculus and first-order logic.

Future work includes the experimentation with the LEO-II on case studies in verification and ontology reasoning.

References

1. Andrews, P.B., Brown, C.E.: TPS: A hybrid automatic-interactive system for developing proofs. J. Applied Logic 4(4), 367–395 (2006)
2. Beeson, M.: Mathematical induction in Otter-Lambda. J. Autom. Reasoning 36(4), 311–344 (2006)

3. Benzmüller, C.: Equality and Extensionality in Automated Higher-Order Theorem Proving. PhD thesis, Saarlandes University (1999)
4. Benzmüller, C., Kohlhase, M.: LEO – a higher-order theorem prover. In: Kirchner, C., Kirchner, H. (eds.) CADE 1998. LNCS (LNAI), vol. 1421, pp. 139–143. Springer, Heidelberg (1998)
5. Benzmüller, C., Paulson, L.: Festschrift in Honour of Peter B. Andrews on his 70th Birthday. In: Exploring Properties of Normal Multimodal Logics in Simple Type Theory with LEO-II. IFCoLog (to appear, 2007)
6. Benzmüller, C., Paulson, L., Theiss, F., Fietzke, A.: Progress report on LEO-II – an automatic theorem prover for higher-order logic. In: TPHOLs 2007 Emerging Trends Proc., Internal Report 364/07, Department of Computer Science, University Kaiserslautern, Germany, pp. 33–48 (2007)
7. Benzmüller, C., Rabe, F., Sutcliffe, G.: THF0 — the core TPTP language for classical higher-order logic. In: Proc (IJCAR 2008). LNCS, vol. 5195. Springer, Heidelberg (2008)
8. Benzmüller, C., Sorge, V., Jamnik, M., Kerber, M.: Experiments with an Agent-Oriented Reasoning System. In: Baader, F., Brewka, G., Eiter, T. (eds.) KI 2001. LNCS (LNAI), vol. 2174, pp. 409–424. Springer, Heidelberg (2001)
9. Benzmüller, C., Sorge, V., Jamnik, M., Kerber, M.: Combined reasoning by automated cooperation. J. Applied Logic (in print) (2008)
10. Bishop, M., Andrews, P.B.: Selectively instantiating definitions. In: Kirchner, C., Kirchner, H. (eds.) CADE 1998. LNCS (LNAI), vol. 1421, pp. 365–380. Springer, Heidelberg (1998)
11. Brown, C.E.: Solving for Set Variables in Higher-Order Theorem Proving. In: Voronkov, A. (ed.) CADE 2002. LNCS (LNAI), vol. 2392, pp. 408–422. Springer, Heidelberg (2002)
12. de Bruijn, N.G.: Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. Indag. Math. 34(5), 381–392 (1972)
13. Gordon, M.J., Melham, T.F.: Introduction to HOL: A Theorem-Proving Environment for Higher-Order Logic. Cambridge University Press, Cambridge (1993)
14. Kerber, M.: On the Representation of Mathematical Concepts and their Translation into First-Order Logic. PhD thesis, Univ. Kaiserslautern, Germany (1992)
15. Klein, L.: Indexing für Terme höherer Stufe. Master's thesis, Saarland Univ. (1997)
16. McCune, W.: Experiments with discrimination-tree indexing and path indexing for term retrieval. J. Automated Reasoning 9(2), 147–167 (1992)
17. Nieuwenhuis, R., Hillenbrand, T., Riazanov, A., Voronkov, A.: On the evaluation of indexing techniques for theorem proving. In: Goré, R.P., Leitsch, A., Nipkow, T. (eds.) IJCAR 2001. LNCS (LNAI), vol. 2083, pp. 257–271. Springer, Heidelberg (2001)
18. Nipkow, T., Paulson, L.C., Wenzel, M.T.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
19. Pientka, B.: Higher-order substitution tree indexing. In: Palamidessi, C. (ed.) ICLP 2003. LNCS, vol. 2916, pp. 377–391. Springer, Heidelberg (2003)
20. Riazanov, A., Voronkov, A.: The design and implementation of Vampire. AICOM 15(2-3), 91–110 (2002)
21. Schulz, S.: E – A Brainiac Theorem Prover. J. AI Communications 15(2/3), 111–126 (2002)
22. Siekmann, J., Benzmüller, C., Autexier, S.: Computer supported mathematics with OMEGA. J. Applied Logic 4(4), 533–559 (2006)

23. Stickel, M.: The path-indexing method for indexing terms. Technical Report 473, Artificial Intelligence Center, SRI International, USA (1989)
24. Theiss, F., Benzmüller, C.: Term indexing for the LEO-II prover. In: Proc. of the 6th International Workshop on the Implementation of Logics. CEUR Workshop Proc., vol. 212 (2006)
25. Weidenbach, C., et al.: Spass version 2.0. In: Voronkov, A. (ed.) CADE 2002. LNCS (LNAI), vol. 2392, pp. 275–279. Springer, Heidelberg (2002)

KeYmaera: A Hybrid Theorem Prover for Hybrid Systems (System Description)*

André Platzer and Jan-David Quesel

University of Oldenburg, Department of Computing Science, Germany
{platzer,quesel}@informatik.uni-oldenburg.de

Abstract. KeYmaera is a hybrid verification tool for hybrid systems that combines deductive, real algebraic, and computer algebraic prover technologies. It is an automated and interactive theorem prover for a natural specification and verification logic for hybrid systems. KeYmaera supports *differential dynamic logic*, which is a real-valued first-order dynamic logic for hybrid programs, a program notation for hybrid automata. For automating the verification process, KeYmaera implements a generalized free-variable sequent calculus and automatic proof strategies that decompose the hybrid system specification symbolically. To overcome the complexity of real arithmetic, we integrate real quantifier elimination following an iterative background closure strategy. Our tool is particularly suitable for verifying parametric hybrid systems and has been used successfully for verifying collision avoidance in case studies from train control and air traffic management.

Keywords: dynamic logic, automated theorem proving, decision procedures, computer algebra, verification of hybrid systems.

1 Introduction

Formal verification becomes more and more important as computerized control systems in safety-critical systems grow significantly in complexity. In many applications, system states, like positions of vehicles, change continuously according to differential equations and are affected by discrete controller decisions. Hybrid systems [7] are a mathematical model for such systems with interacting discrete and continuous dynamics. Model checkers [7,5] verify correctness properties by exploring the state space exhaustively, which provides a good mechanism to find bugs or concrete counterexamples for specifications. Unfortunately, the state space of hybrid systems is uncountably infinite and cannot be partitioned into finitely many relevant regions for deciding reachability [7].

As deductive methods [4,2,8,1] are known for being capable of dealing with infinite domains, we choose a proof-based approach. We present the verification tool KeYmaera that uses a combination of automated theorem proving (for

* This research was partially supported by the German Research Council (DFG) in the Transregional Collaborative Research Center SFB/TR 14 AVACS.

symbolically decomposing and executing system models), real quantifier elimination [3] (for handling the arithmetic of hybrid systems), and symbolic computations in computer algebra systems (for handling differential equations of continuous evolutions). As the central concept, our tool implements an axiomatization of the transition behavior of hybrid systems in the form of the sequent calculus for the differential dynamic logic d$\mathcal{L}$ [12,13]. KeYmaera provides proof strategies that automate the verification process to a large extent. In several realistic applications, the proof construction is even completely automatic, e.g., for proving collision avoidance of trains or aircraft.

In addition, theorem proving in combination with the equivalence-transformations of quantifier elimination enables us to verify highly parametrized hybrid systems and even to discover safety-critical parameter constraints. The traceability gained by the deductive symbolic system decomposition enables the user to use his system knowledge for projecting the obtained constraints on the free parameters of the system to the relevant cases. Since the underlying logic d$\mathcal{L}$ and its compositional proof calculus are natural and intuitive, even computationally intractable problems can be verified with selective user guidance in KeYmaera.

In this paper, we describe the theorem prover KeYmaera and the various techniques that it combines for verifying hybrid systems. KeYmaera consists of ca. 186,000 lines of Java code and 141 optimized proof rules, including rules for symbolic decomposition, propositional logic, first-order logic, and simplification.

2 KeYmaera Verification Tool for Hybrid Systems

KeYmaera is a deductive verification tool for hybrid systems. We have implemented KeYmaera as a combination of the deductive theorem prover KeY [2] with the computer algebra system Mathematica, see Fig. 1. KeY is an interactive theorem prover with a user-friendly graphical interface for proving correctness properties of Java programs. We generalize KeY from discrete systems to hybrid systems by adding support for the differential dynamic logic d$\mathcal{L}$ [12,13], which is a dynamic logic [6] that provides a natural way to formalize properties of the states reachable by following the dynamics of hybrid systems. With this, KeYmaera can prove correctness, safety, controllability, reactivity, and liveness properties of hybrid systems.

In discrete KeY, rule applications are comparably fast, but in KeYmaera, proof rules that use decision procedures for real arithmetic can require a substantial amount of time to produce a result. To overcome this, we have implemented

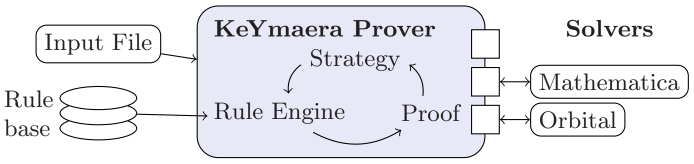
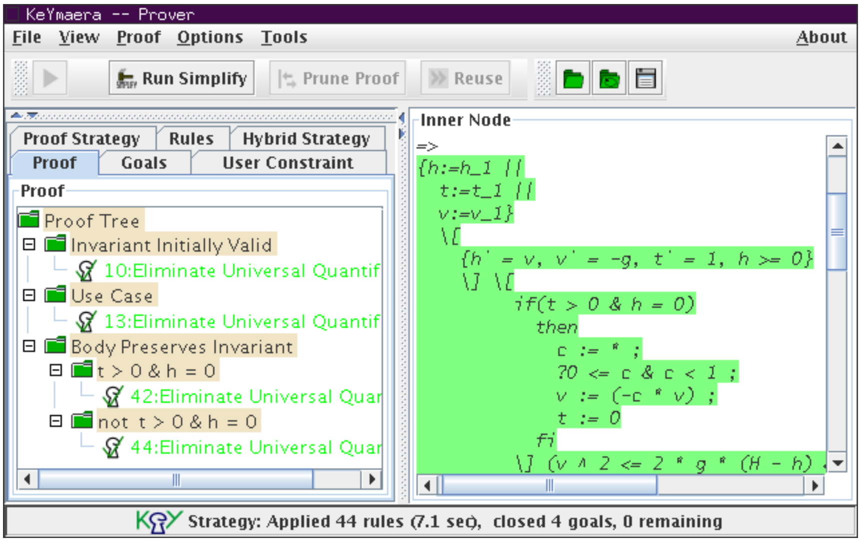


Fig. 1. Architecture and plug-in structure of the KeYmaera Prover



```
Proof Tree
⊟ Invariant Initially Valid
    10:Eliminate Universal Quantif
⊟ Use Case
    13:Eliminate Universal Quantif
⊟ Body Preserves Invariant
  ⊟ t > 0 & h = 0
      42:Eliminate Universal Quar
  ⊟ not t > 0 & h = 0
      44:Eliminate Universal Quar
```

```
=>
{h:=h_1 ||
  t:=t_1 ||
  v:=v_1}
  \[
    {h' = v, v' = -g, t' = 1, h >= 0}
  \] \[
      if(t > 0 & h = 0)
        then
          c := * ;
          ?0 <= c & c < 1 ;
          v := (-c * v) ;
          t := 0
        fi
    \] (v ^ 2 <= 2 * g * (H - h)
```

Strategy: Applied 44 rules (7.1 sec), closed 4 goals, 0 remaining

Fig. 2. Screenshot of the KeYmaera user interface

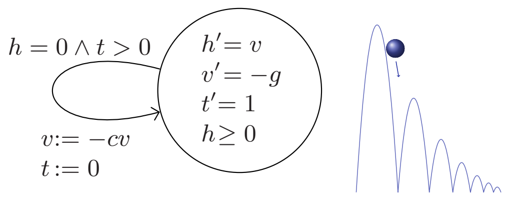
new automatic proof strategies for the hybrid case that navigate among computationally expensive rule applications.

We have implemented a plug-in architecture for integrating multiple implementations of decision procedures for the different fields of arithmetic handling, cf. Fig. 1. We integrate arithmetical simplification and real quantifier elimination support by interfacing Mathematica. Symbolic solutions of differential equations, which can be used for handling continuous dynamics, are obtained either from Mathematica or Orbital, a math library for Java developed by the first author.

3 Hybrid Systems, Hybrid Automata, and Hybrid Programs

Hybrid systems [7] are mathematical models for systems with interacting continuous and discrete state transitions. The standard description language for specifying the operational behavior of hybrid systems is that of *hybrid automata* [7]. A hybrid automaton is a finite automaton with real variables that evolve continuously in the automaton locations as specified by differential equations. Additionally, state variables can occur in transition guards and transitions can change the values of the variables. The graph notation of hybrid automata is not compositional, e.g., it is not sufficient to prove properties separately for each location to infer a global system property, as the transition effects have to be taken into account.

Instead, we use a program notation for hybrid automata which is designed to have a compositional semantics that we exploit for verifying systems by symbolic decomposition. *Hybrid programs* [12,13] are an extension of discrete regular programs [6] by continuous evolutions. An overview of the syntax and



```
Ball ≡ (
  $(h' = v, v' = -g, t' = 1, h \geq 0)$;
  if $(h = 0 \ \land \ t > 0)$ then
    $c := *$; $?0 \leq c < 1$;   // add-on
    $v := -cv$; $t := 0$
  fi$)^*$
```

Fig. 3. Hybrid automaton of a bouncing ball and corresponding hybrid program

informal semantics of hybrid programs is given in Tab. 1 (where F is a formula of first-order real arithmetic). Hybrid automata can be embedded into hybrid programs [13].

Example 1. Consider the well-known bouncing ball example. A ball falls from height h and bounces back from the ground ($h = 0$) after an elastic deformation. The current speed of the ball is denoted by v, and t is a clock measuring the falling time. We assume an arbitrary positive gravity force g and that the ball loses energy according to a damping factor $0 \leq c < 1$. Fig. 3 depicts the hybrid automaton, an illustration of the system dynamics, and the representation of the system as a hybrid program. However, the automaton still enforces infinite bouncing. In reality, the ball can come to a standstill when its remaining kinetic energy is insufficient. To model this phenomenon without the need to have a precise physical model for all involved forces, we allow for the damping factor to change at each bounce. Line 4 of the hybrid program in Fig. 3 represents a corresponding uncountably infinite nondeterministic choice for c, which is beyond the modelling capabilities of hybrid automata.

4 Syntax and Semantics of Differential Dynamic Logic

For characterizing states of hybrid systems, the foundation of the specification and verification logic of KeYmaera is first-order logic over real arithmetic. For

Table 1. Statements of hybrid programs

Statement	Effect
$\alpha;\ \beta$	sequential composition, first performs α and then β afterwards
$\alpha\ \cup\ \beta$	nondeterministic choice, following either α or β
α^*	nondeterministic repetition, repeating α $n \geq 0$ times
$x := \theta$	discrete assignment of the value of term θ to variable x (jump)
$x := *$	nondeterministic assignment of an arbitrary real number to x
$(x_1' = \theta_1, \ldots, x_n' = \theta_n, F)$	continuous evolution of x_i along differential equation system $x_i' = \theta_i$, restricted to maximum domain or invariant region F
$?F$	check if formula F holds at current state, abort otherwise
if(F) then α	perform α if F holds, do nothing otherwise
if(F) then α else β	perform α if F holds, perform β otherwise

expressing correctness statements about hybrid systems, this foundation is extended with parametrized modal operators $[\alpha]$ and $\langle\alpha\rangle$, for each hybrid program α. The resulting specification and verification logic is called *differential dynamic logic* $\mathsf{d}\mathcal{L}$ [12,13].

As is typical in dynamic logic, $\mathsf{d}\mathcal{L}$ integrates operational system models and formulas within a single language. The $\mathsf{d}\mathcal{L}$ formulas are generated by the following EBNF grammar (where $\sim \in \{<, \leq, =, \geq, >\}$ and θ_1, θ_2 are arithmetic expressions in $+, -, \cdot, /$ over the reals):

$$\phi ::= \theta_1 \sim \theta_2 \mid \neg\phi \mid \phi \wedge \psi \mid \forall x \phi \mid \exists x \phi \mid [\alpha]\phi \mid \langle\alpha\rangle\phi$$

The modal operators refer to states reachable by the hybrid program α and can be placed in front of any formula. Formula $[\alpha]\phi$ expresses that all states reachable by the hybrid program α satisfy formula ϕ (safety). Formula $\langle\alpha\rangle\phi$ expresses that there is a state reachable by the hybrid program α that satisfies formula ϕ.

Example 2. The ball looses energy at every bounce, thus the ball never bounces higher than the initial height. This can be expressed by the safety property $0 \leq h \leq H$, where H denotes the initial energy level, i.e., the initial height if $v = 0$. As a simple specification, which we generalize to the general parametric case later on, we can verify the following property using KeYmaera:

$$(h = H \wedge v = 0 \wedge 0 \leq H \leq 4 \wedge 0 < g \leq 2) \rightarrow [\mathtt{Ball}]\,(0 \leq h \leq H) \qquad (1)$$

This specification follows the pattern of Hoare-triples. It expresses that the bouncing ball, when started in initial state $h = H$ etc. always respects $0 \leq h \leq H$.

The semantics of $\mathsf{d}\mathcal{L}$ is a Kripke semantics [12,13] where states correspond to states of the hybrid system, i.e., assignments of real values to all system variables.

5 Verification by Symbolic Decomposition

Exploiting the compositional semantics of $\mathsf{d}\mathcal{L}$, KeYmaera verifies properties of hybrid programs by proving corresponding properties of their parts in a sequent calculus [12,13]. For instance, the proof for specification (1) splits into a case where the if-statement takes effect and one where its condition gives false so that its body is skipped:

$$\frac{h = 0 \vdash [v := -cv]h \leq H \quad h \neq 0 \vdash h \leq H}{\vdash [\text{if } h = 0 \ \text{ then } v := -cv \text{ fi}]h \leq H} \quad \text{by } \mathsf{d}\mathcal{L} \text{ rule} \quad \frac{F \vdash [\alpha]\phi \quad \neg F \vdash \phi}{\vdash [\text{if } F \text{ then } \alpha \text{ fi}]\phi}$$

6 Real Arithmetic and Computer Algebra

One challenge when verifying hybrid systems is the handling of intricate arithmetic resulting from continuous evolution along differential equations. If the computer algebra system does not find a polynomial solution, we handle differential equations by their local dynamics using differential induction [11]. Otherwise, we substitute all occurrences of the evolution variables by their solutions at position τ. There, the fresh variable τ represents the evolution duration and is universally quantified for proving $[\alpha]\phi$ and existentially quantified for $\langle\alpha\rangle\phi$. Additionally, for all times in between 0 and τ the invariant must hold. Using the solution of the differential equation, the property $h \leq H$ in the proof of Example 1 yields:

$$\ldots \vdash \forall\tau{\geq}0\left((\forall\tilde{\tau}\,(0{\leq}\tilde{\tau}{\leq}\tau \rightarrow \frac{-g}{2}\tilde{\tau}^2 + \tilde{\tau}v + h \geq 0)) \rightarrow \frac{-g}{2}\tau^2 + \tau v + h \leq H\right) \quad (2)$$

The inner quantifier checks if the invariant region $h \geq 0$ is respected at all times during the evolution. The symbolic decomposition rules of d$\mathcal{L}$ result in quantified arithmetical formulas like (2). KeYmaera handles them using real quantifier elimination [3] as a decision procedure for real arithmetic, which is provided, e.g., using a seamless integration of KeYmaera with Mathematica.

7 Automation and Iterative Background Closure

KeYmaera implements a verification algorithm [10] that decomposes d$\mathcal{L}$ formulas recursively in the d$\mathcal{L}$ calculus. For the resulting arithmetical formulas, real quantifier elimination can be intractable in theory and practice. Experiments show that neither eager nor lazy calls of background solvers are feasible [10]. Due to the doubly exponential complexity of real quantifier elimination, eager calls seldom terminate in practice. For lazy calls, instead, the proof splits into multiple sub-problems, which can be equally computationally expensive.

To overcome the complexity pitfalls of quantifier elimination and to scale to real-world application scenarios, we implement an *iterative background closure* strategy [10] that interleaves background solver calls with deductive d$\mathcal{L}$ rules. The basic idea is to interrupt background solvers after a timeout and only restart them after a d$\mathcal{L}$ rule has split the proof. In addition, we increase timeouts for sub-goals according to a simple exponential scheme, see Fig. 4. The effect is that KeYmaera avoids splitting goals in the average case but is still able to split cases with prohibitive computational cost along their first-order and propositional structure.

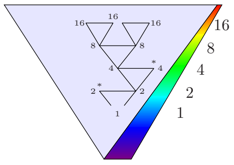


Fig. 4. Incremental timeouts in proof search space

8 Parameter Discovery

Required parameter constraints can be discovered in KeYmaera by selecting the appropriate parts obtained by symbolic decomposition and transformation by quantifier elimination. Essentially, equivalence-transformations of quantifier elimination yield equivalent parameter constraints when a proof does not succeed, which can be exploited for parameter discovery. See [12,13] for details.

Example 3. To obtain a fully parametric invariant for Example 2, properties for isolated modes like $[h'' = -g]h \leq H$ can be analyzed in KeYmaera. By selecting the relevant constraints from the resulting formula we obtain the invariant

$$v^2 \leq 2g(H-h) \wedge h \geq 0$$

which will be used for verifying the nondeterministic repetition of the system in Fig. 3. Assuming the invariant to hold in the initial state, we obtain a general parametric specification for the bouncing ball which is provable using KeYmaera:

$$(v^2 \leq 2g(H-h) \wedge h \geq 0 \wedge g > 0 \wedge H \geq 0) \rightarrow [\texttt{Ball}]\,(0 \leq h \leq H)$$

9 Applications

KeYmaera can be used for verifying hybrid systems even in the presence of parameters in the system dynamics. Its flexible specification logic d$\mathcal{L}$ can be used for discovering the required parameter constraints. We have verified several properties of the European Train Control System (ETCS) successfully in KeYmaera, including safety, liveness, controllability, and reactivity, thereby entailing collision freedom [14]. In addition, collision avoidance has been verified for roundabout maneuvers in air traffic management [16], which involve challenging continuous dynamics with trigonometric functions.

10 Related Work

Davoren and Nerode [4] outline other uses of logic in hybrid systems. Theorem provers have been used for verifying hybrid systems in STeP [8] or PVS [1]. However, they do not use a genuine logic for hybrid systems but compile prespecified invariants of hybrid automata into an overall verification condition. Further, by using background solvers and iterative background closure strategies, we obtain a larger degree of automation than interactive proving in STeP [8] or higher-order logic [1]. VSE-II [9] uses discrete approximations of hybrid automata for verification. In contrast, KeYmaera respects the full continuous-time semantics of hybrid systems. PHAVer [5] is a model checker primarily for linear hybrid automata. CheckMate [15] supports more complex continuous dynamics, but still requires initial states and switching surfaces to be linear. KeYmaera supports nonlinear constraints on the system parameters as required for train applications or even the parametric bouncing ball.

References

1. Ábrahám-Mumm, E., Steffen, M., Hannemann, U.: Verification of hybrid systems: Formalization and proof rules in PVS. In: ICECCS, pp. 48–57. IEEE Computer, Los Alamitos (2001)
2. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007)

3. Collins, G.E., Hong, H.: Partial cylindrical algebraic decomposition for quantifier elimination. J. Symb. Comput. 12(3), 299–328 (1991)
4. Davoren, J.M., Nerode, A.: Logics for hybrid systems. IEEE 88(7) (July 2000)
5. Frehse, G.: PHAVer: Algorithmic verification of hybrid systems past HyTech. In: Morari, M., Thiele, L. (eds.) HSCC 2005. LNCS, vol. 3414, pp. 258–273. Springer, Heidelberg (2005)
6. Harel, D., Kozen, D., Tiuryn, J.: Dynamic logic. MIT Press, Cambridge (2000)
7. Henzinger, T.A.: The theory of hybrid automata. In: LICS, pp. 278–292. IEEE Computer Society, Los Alamitos (1996)
8. Manna, Z., Sipma, H.: Deductive verification of hybrid systems using STeP. In: Henzinger, T.A., Sastry, S.S. (eds.) HSCC 1998. LNCS, vol. 1386. Springer, Heidelberg (1998)
9. Nonnengart, A., Rock, G., Stephan, W.: Using hybrid automata to express realtime properties in VSE-II. In: Russell, I., Kolen, J.F. (eds.) FLAIRS. AAAI Press, Menlo Park (2001)
10. Platzer, A.: Combining deduction and algebraic constraints for hybrid system analysis. In: Beckert, B. (ed.) VERIFY 2007 at CADE 2007, CEUR-WS.org (2007)
11. Platzer, A.: Differential algebraic dynamic logic for differential algebraic programs. (submitted, 2007)
12. Platzer, A.: Differential dynamic logic for verifying parametric hybrid systems. In: Olivetti, N. (ed.) TABLEAUX 2007. LNCS (LNAI), vol. 4548, pp. 216–232. Springer, Heidelberg (2007)
13. Platzer, A.: Differential dynamic logic for hybrid systems. J. Autom. Reasoning (to appear, 2008)
14. Platzer, A., Quesel, J.D.: Logical verification and systematic parametric analysis in train control. In: Egerstedt, M., Mishra, B. (eds.) HSCC. LNCS, Springer, Heidelberg (2008)
15. Silva, B.I., Richeson, K., Krogh, B.H., Chutinan, A.: Modeling and verification of hybrid dynamical system using CheckMate. In: ADPM 2000: 4th International Conference on Automation of Mixed Processes: Hybrid Dynamic Systems (2000)
16. Tomlin, C., Pappas, G.J., Sastry, S.: Conflict resolution for air traffic management: a study in multi-agent hybrid systems. IEEE T. Automat. Contr. 43(4) (1998)

The Complexity of Conjunctive Query Answering in Expressive Description Logics

Carsten Lutz

Institut für Theoretische Informatik
TU Dresden, Germany
lutz@tcs.inf.tu-dresden.de

Abstract. Conjunctive query answering plays a prominent role in applications of description logics (DLs) that involve instance data, but its exact complexity was a long-standing open problem. We determine the complexity of conjunctive query answering in expressive DLs between $\mathcal{ALC}$ and $\mathcal{SHIQ}$, and thus settle the problem. In a nutshell, we show that conjunctive query answering is 2ExpTime-complete in the presence of inverse roles, and only ExpTime-complete without them.

1 Introduction

Description logics (DLs) originated in the late 1970ies as knowledge representation (KR) formalisms, and nowadays play an important role as ontology languages [1]. Traditionally, DLs are used for the representation of and reasoning about the conceptual modeling of an application domain. Most KR applications of DLs are of this kind, and also the majority of ontologies focusses on conceptual modeling. In contrast, more recent applications of DLs additionally involve (potentially large amounts of) instance data. In particular, instance data plays an important role when using DL ontologies for data-integration and in ontology-mediated data access.

In DLs, a TBox is used to represent conceptual information, and instance data is stored in the ABox. Consequently, traditional DL research has mainly concentrated on TBox reasoning, where the main reasoning services are subsumption and satisfiability. In the presence of ABoxes, additional reasoning services are required to query the instance data. The most basic such service is *instance retrieval*, i.e., to return all certain answers to a query that has the form of a DL concept. Instance retrieval can be viewed as a well-behaved generalization of subsumption and satisfiability: it is usually possible to adapt algorithms in a straightforward way, and the computational complexity coincides in almost all cases (but see [20] for an exception). A more powerful way to query ABoxes is *conjunctive query answering*, as first studied in the context of DLs by Calvanese et al. in 1998 [2]. Roughy speaking, conjunctive query answering generalizes instance retrieval by admitting also queries whose relational structure is not tree-shaped. This generalization is both natural and useful because the relational structure of ABoxes is usually not tree-shaped either.

Conjunctive queries have been studied extensively in the DL literature, see for example [2,3,4,6,7,9,10,11,17,21]. In contrast to the case of instance retrieval, developing algorithms for conjunctive query answering requires novel techniques. In particular, all hitherto known algorithms for conjunctive query answering in the basic propositionally closed DL $\mathcal{ALC}$ and its extensions require double exponential time. In contrast, subsumption, satisfiability, and instance checking (the decision problem corresponding to instance retrieval) are ExpTime-complete even in the expressive DL $\mathcal{SHIQ}$, which is a popular extension of $\mathcal{ALC}$ [8]. It follows that, in DLs between $\mathcal{ALC}$ and $\mathcal{SHIQ}$, the complexity of conjunctive query entailment (the decision problem corresponding to conjunctive query answering) is between ExpTime and 2ExpTime. However, the exact complexity of this important problem has been open for a long time. In particular, it was unclear whether the generalization of instance retrieval to conjunctive query answering comes with an increase in computational complexity.

In this paper, we settle the problem and determine the exact complexity of conjunctive query entailment in DLs between $\mathcal{ALC}$ and $\mathcal{SHIQ}$. More precisely, we show that

(1) Conjunctive query entailment in $\mathcal{ALCI}$, the extension of $\mathcal{ALC}$ with inverse roles, is 2ExpTime-hard. With the upper bound from [7], conjunctive query answering is thus 2ExpTime-complete for any DL between $\mathcal{ALCI}$ and $\mathcal{SHIQ}$.

(2) Conjunctive query entailment in $\mathcal{SHQ}$ is in ExpTime. With the ExpTime lower bound for instance checking in $\mathcal{ALC}$, conjunctive query entailment is thus ExpTime-complete for any DL between $\mathcal{ALC}$ and $\mathcal{SHQ}$.

In short, conjunctive query entailment is one exponential harder than instance checking in the presence of inverse roles, but not without them. Result (2) was proved independently and in parallel for the DL $\mathcal{ALCH}$ in [18], and generalized to also include transitive roles (under some restrictions) in [19].

We also consider the special case of conjunctive query entailment where the query is *rooted*, i.e., it is connected and contains at least one answer variable. We prove matching lower and upper bounds to show that

(3) Rooted conjunctive query entailment is NExpTime-complete for any DL between $\mathcal{ALCI}$ and $\mathcal{SHIQ}$.

Thus, rootedness reduces the complexity of query entailment in the presence of inverse roles (but not without them). In the upper bounds of (2) and (3), we disallow transitive and other so-called non-simple roles in the query. We also show that rooted conjunctive query entailment in $\mathcal{ALCI}$ with transitive roles becomes 2ExpTime-complete if transitive roles are admitted in the query.

This paper is organized as follows. In Section 2, we briefly review some preliminaries. We then establish the lower bounds, starting with the NExpTime one of (3) in Section 3. The 2ExpTime lower bound of (1) builds on that, but we have to confine ourselves to a brief sketch in Section 4. This section also establishes 2ExpTime-hardness of $\mathcal{ALCI}$ with transitive roles in the query. In Section 5, we prove the ExpTime upper bound of (2). In Section 6, we give some further discussion of transitive roles in the query. This paper is based on the workshop papers [15] and [16].

2 Preliminaries

We assume standard notation for the syntax and semantics of $\mathcal{SHIQ}$ knowledge bases [8]. In particular, $\mathsf{N_C}$, $\mathsf{N_R}$, and $\mathsf{N_I}$ are countably infinite and disjoint sets of *concept names*, *role names*, and *individual names*. A *TBox* is a set of concept inclusions $C \sqsubseteq D$, role inclusions $r \sqsubseteq s$, and transitivity statements $\mathsf{Trans}(r)$, and a *knowledge base (KB)* is a pair $(\mathcal{T}, \mathcal{A})$ consisting of a TBox $\mathcal{T}$ and an ABox $\mathcal{A}$. We write $\mathcal{K} \models s \sqsubseteq r$ if the role inclusion $s \sqsubseteq r$ is true in all models of $\mathcal{K}$, and similarly for $\mathcal{K} \models \mathsf{Trans}(r)$. It is easy to see and well-known that "$\mathcal{K} \models s \sqsubseteq r$" and "$\mathcal{K} \models \mathsf{Trans}(r)$" are decidable in polytime [8]. As usual, a role is called *simple* if there is no role s such that $\mathcal{K} \models s \sqsubseteq r$, and $\mathcal{K} \models \mathsf{Trans}(s)$. We write $\mathsf{Ind}(\mathcal{A})$ to denote the set of all individual names in an ABox $\mathcal{A}$. Throughout the paper, the number n inside number restrictions $({\geq}\, n\, r\, C)$ and $({\leq}\, n\, r\, C)$ is assumed to be coded in binary. $\mathcal{ALC}$ is the fragment of $\mathcal{SHIQ}$ that disallows role hierarchies, transitive roles, inverse roles, and number restrictions.

Let $\mathsf{N_V}$ be a countably infinite set of *variables*. An *atom* is an expression $C(v)$ or $r(v, v')$, where C is a $\mathcal{SHIQ}$ concept, r is a simple (but possibly inverse) role, and $v, v' \in \mathsf{N_V}$. A *conjunctive query* q is a finite set of atoms. We use $\mathsf{Var}(q)$ to denote the set of variables occurring in the query q. For each query q, the set $\mathsf{Var}(q)$ is partitioned into *answer variables* and (existentially) *quantified variables*. Let $\mathcal{A}$ be an ABox, $\mathcal{I}$ a model of $\mathcal{A}$, q a conjunctive query, and $\pi : \mathsf{Var}(q) \to \Delta^{\mathcal{I}}$ a total function. such that for every answer variable $v \in \mathsf{Var}(q)$, there is an $a \in \mathsf{N_I}$ such that $\pi(v) = a^{\mathcal{I}}$. We write $\mathcal{I} \models^{\pi} C(v)$ if $\pi(v) \in C^{\mathcal{I}}$ and $\mathcal{I} \models^{\pi} r(v, v')$ if $(\pi(v), \pi(v')) \in r^{\mathcal{I}}$. If $\mathcal{I} \models^{\pi} at$ for all $at \in q$, we write $\mathcal{I} \models^{\pi} q$ and call π a *match* for $\mathcal{I}$ and q. We say that $\mathcal{I}$ *satisfies* q and write $\mathcal{I} \models q$ if there is a match π for $\mathcal{I}$ and q. If $\mathcal{I} \models q$ for all models $\mathcal{I}$ of a KB $\mathcal{K}$, we write $\mathcal{K} \models q$ and say that $\mathcal{K}$ *entails* q.

The *query entailment problem* is, given a knowledge base $\mathcal{K}$ and a query q, to decide whether $\mathcal{K} \models q$. This is the decision problem corresponding to query answering, see e.g. [7] for details. Observe that we do not admit the use of individual constants in conjunctive queries. This assumption is only for simplicity, as such constants can easily be simulated by introducing additional concept names [7]. We speak of *rooted query entailment* when the query q is *rooted*, i.e., when q is connected and contains at least one answer variable.

3 Rooted Query Entailment in $\mathcal{ALCI}$ and $\mathcal{SHIQ}$

$\mathcal{ALCI}$ is the extension of $\mathcal{ALC}$ with inverse roles, and thus a fragment of $\mathcal{SHIQ}$. The aim of this section is to show that rooted query entailment in $\mathcal{ALCI}$ is NExpTime-complete in all DLs between $\mathcal{ALCI}$ and $\mathcal{SHIQ}$. To comply with space limitations, we concentrate on the lower bound. It applies even to the case where TBoxes are empty.

Let $\mathcal{ALC}^{\mathsf{rs}}$ be the variation of $\mathcal{ALC}$ in which all roles are interpreted as reflexive and symmetric relations. Our proof of the NExpTime lower bound proceeds by first polynomially reducing rooted query entailment in $\mathcal{ALC}^{\mathsf{rs}}$ w.r.t. the empty

TBox to rooted query entailment in $\mathcal{ALCI}$ w.r.t. the empty TBox. Then, we prove co-NExpTime-hardness of rooted query entailment in $\mathcal{ALC}^{\mathsf{rs}}$. Regarding the first step, the idea is to replace each symmetric role s with the composition of r^- and r, with r a role of $\mathcal{ALCI}$. Although r is not interpreted in a symmetric relation, the composition of r^- and r is clearly symmetric. To achieve reflexivity, we ensure that $\exists r^-.\top$ is satisfied by all relevant individuals and for all relevant roles r. Then, every domain element can reach itself by first travelling r^- and then r, which corresponds to a reflexive s-loop. Since we are working without TBoxes and thus cannot use statements such as $\top \sqsubseteq \exists r^-.\top$, a careful manipulation of the ABox and query is needed. Details are given in [15].

Before we prove co-NExpTime-hardness of rooted query entailment in $\mathcal{ALC}^{\mathsf{rs}}$ with empty TBoxes, we discuss a preliminary. An interpretation $\mathcal{I}$ of $\mathcal{ALC}^{\mathsf{rs}}$ is *tree-shaped* if there is a bijection f from $\Delta^{\mathcal{I}}$ into the set of nodes of a finite undirected tree (V, E) such that $(d, e) \in s^{\mathcal{I}}$, for some role name s, implies that $d = e$ or $\{f(d), f(e)\} \in E$. The proof of the following result is standard, using unravelling.

Lemma 1. *If $\mathcal{A}$ is an $\mathcal{ALC}^{\mathsf{rs}}$-ABox and q a conjunctive query, then $\mathcal{A} \not\models q$ implies that there is a tree-shaped model $\mathcal{I}$ of $\mathcal{A}$ such that $\mathcal{I} \not\models q$.*

Thus, we can concentrate on tree-shaped interpretations throughout the proof. We now give a reduction from a NExpTime-complete variant of the tiling problem to rooted query *non*-entailment in $\mathcal{ALC}^{\mathsf{rs}}$.

Definition 1 (Domino System). *A* domino system $\mathfrak{D}$ *is a triple (T, H, V), where $T = \{0, 1, \ldots, k-1\}$, $k \geq 0$, is a finite set of* tile types *and $H, V \subseteq T \times T$ represent the* horizontal and vertical matching conditions. *Let $\mathfrak{D}$ be a domino system and $c = c_0, \ldots, c_{n-1}$ an* initial condition, *i.e. an n-tuple of tile types. A mapping $\tau : \{0, \ldots, 2^{n+1}-1\} \times \{0, \ldots, 2^{n+1}-1\} \to T$ is a* solution *for $\mathfrak{D}$ and c iff for all $x, y < 2^{n+1}$, the following holds (where $\oplus_i$ denotes addition modulo i): (i) if $\tau(x, y) = t$ and $\tau(x \oplus_{2^{n+1}} 1, y) = t'$, then $(t, t') \in H$; (ii) if $\tau(x, y) = t$ and $\tau(x, y \oplus_{2^{n+1}} 1) = t'$, then $(t, t') \in V$; (iii) $\tau(i, 0) = c_i$ for $i < n$.*

For NExpTime-hardness of this problem see, e.g., Corollary 4.15 in [13]. We show how to translate a given domino system $\mathfrak{D}$ and initial condition $c = c_0 \cdots c_{n-1}$ into an ABox $\mathcal{A}_{\mathfrak{D},c}$ and query $q_{\mathfrak{D},c}$ such that each tree-shaped model $\mathcal{I}$ of $\mathcal{A}_{\mathfrak{D},c}$ that satisfies $\mathcal{I} \not\models q_{\mathfrak{D},c}$ encodes a solution to $\mathfrak{D}$ and c, and conversely, each solution to $\mathfrak{D}$ and c gives rise to a (tree-shaped) model of $\mathcal{A}_{\mathfrak{D},c}$ with $\mathcal{I} \not\models q_{\mathfrak{D},c}$. The ABox $\mathcal{A}_{\mathfrak{D},c}$ contains only the assertion $C_{\mathfrak{D},c}(a)$, with $C_{\mathfrak{D},c}$ a conjunction $C^1_{\mathfrak{D},c} \sqcap \cdots \sqcap C^7_{\mathfrak{D},c}$ whose conjuncts we define in the following. For convenience, let $m = 2n+2$. The purpose of the first conjunct $C^1_{\mathfrak{D},1}$ is to enforce a binary tree of depth m whose leaves are labelled with the numbers $0, \ldots, 2^m - 1$ of a binary counter implemented by the concept names $A_0, \ldots, A_{m-1}$. We use concept names $L_0, \ldots, L_m$ to distinguish the different levels of the tree. This is necessary because we work with reflexive and symmetric roles. In the following $\forall s^i.C$ denotes the i-fold nesting $\forall s. \cdots \forall s.C$. In particular, $\forall s^0.C$ is C.

$$\begin{aligned} C^1_{\mathfrak{D},c} := L_0 \sqcap \bigsqcap_{i<m} \forall s^i.\big(L_i \rightarrow \big(\exists s.(L_{i+1} \sqcap A_i) \sqcap \exists s.(L_{i+1} \sqcap \neg A_i)\big)\big) \sqcap \\ \bigsqcap_{i<m} \forall s^i. \bigsqcap_{j<i} \big((L_i \sqcap A_j) \rightarrow \forall s.(L_{i+1} \rightarrow A_j) \sqcap \\ (L_i \sqcap \neg A_j) \rightarrow \forall s.(L_{i+1} \rightarrow \neg A_j)\big) \end{aligned}$$

From now on, leafs in this tree are called L_m-nodes. Each L_m-node corresponds to a position in the $2^{n+1} \times 2^{n+1}$-grid that we have to tile: the counter A_x realized by the concept names $A_0, \dots, A_n$ binarily encodes the horizontal position, and the counter A_y realized by $A_{n+1}, \dots, A_m$ encodes the vertical position. We now extend the tree with some additional nodes. Every L_m-node gets three successor nodes labelled with F, and each of these F-nodes has a successor node labelled G. To distinguish the three different G-nodes below each L_m-node, we additionally label them with the concept names G_1, G_2, G_3.

$$C^2_{\mathfrak{D},c} := \forall s^m.\big(L_m \rightarrow \big(\bigsqcap_{1 \leq i \leq 3} \exists s.(F \sqcap \exists s.(G \sqcap G_i))\big)\big)$$

We want that each G_1-node represents the grid position identified by its ancestor L_m-node, the sibling G_2 node represents the horizontal neighbor position in the grid, and the sibling G_3-node represents the vertical neighbor.

$$\begin{aligned} C^3_{\mathfrak{D},c} := \forall s^m.\big(L_m \rightarrow \big(& \bigsqcap_{i \leq n} \big((A_i \rightarrow \forall s^2.(G_1 \sqcup G_3 \rightarrow A_i)) \sqcap \\ & \quad (\neg A_i \rightarrow \forall s^2.(G_1 \sqcup G_3 \rightarrow \neg A_i))\big) \sqcap \\ & \bigsqcap_{n<i<m} \big((A_i \rightarrow \forall s^2.(G_1 \sqcup G_2 \rightarrow A_i)) \sqcap \\ & \quad (\neg A_i \rightarrow \forall s^2.(G_1 \sqcup G_2 \rightarrow \neg A_i))\big) \sqcap \\ & E_2 \sqcap E_3 \;\big)\big) \end{aligned}$$

where E_2 is an $\mathcal{ALC}$-concept ensuring that the A_x value at each G_2-node is obtained from the A_x-value of its G-node ancestor by incrementing modulo 2^{n+1}; similarly, E_3 expresses that the A_y value at each G_3-node is obtained from the A_y-value of its G-node ancestor by incrementing modulo 2^{n+1}. It is not hard to work out the details of these concepts, see e.g. [14] for more details. The *grid representation* that we have enforced is shown in Figure 1. To represent tiles, we introduce a concept name D_i for each $i \in T$. It is now easy to define concepts $C^4_{\mathfrak{D},c}$ and $C^5_{\mathfrak{D},c}$ which enforce that every G-node is labeled with exactly one tile type, and that the initial condition is satisfied—details are left to the reader. To enforce the matching conditions, we proceed in two steps. First we ensure that they are satisfied locally, i.e., among the three G-nodes below each L_m-node:

$$\begin{aligned} C^6_{\mathfrak{D},c} := \forall s^{m+2}.\big(L_m \rightarrow \big(& \bigsqcap_{i \in T} \big(\exists s^2.(G_1 \sqcap D_i) \rightarrow \forall s^2.(G_2 \rightarrow \bigsqcup_{(i,j) \in H} D_j)\big) \sqcap \\ & \bigsqcap_{i \in T} \big(\exists s^2.(G_1 \sqcap D_i) \rightarrow \forall s^2.(G_3 \rightarrow \bigsqcup_{(i,j) \in V} D_j))\big)\big) \end{aligned}$$

Second, we enforce the following condition, which together with local satisfaction of the matching conditions ensures their global satisfaction:

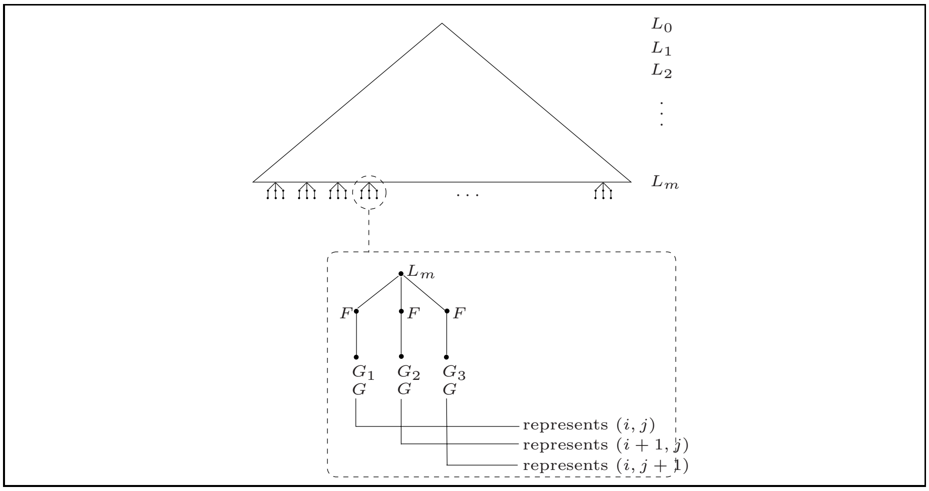


Fig. 1. The structure encoding the $2^{n+1} \times 2^{n+1}$-grid

(∗) if the A_x and A_y-values of two G-nodes coincide, then their tile types coincide.

In (∗), a G-node can by any of a G_1-, G_2-, or G_3-node. To enforce (∗), we use the query. Before we give details, let us finish the definition of the concept $C_{\mathfrak{D},c}$. The last conjunct $C^7_{\mathfrak{D},c}$ enforces two technical conditions that will be explained later: if d is an F-node and e its G-node successor, then

T1 d satisfies A_i iff e satisfies $\neg A_i$, for all $i < m$;
T2 if d satisfies D_j, then e satisfies $D_0, \ldots, D_{j-1}, \neg D_j, D_{j+1}, \ldots, D_{k-1}$, for all $j < k$.

Details of $C^7_{\mathfrak{D},c}$ are left to the reader.

We now construct the query $q_{\mathfrak{D},c}$ that does *not* match the grid representation iff (∗) is satisfied. In other words, $q_{\mathfrak{D},c}$ matches the grid representation iff there are two G-nodes that agree on the value of the counters A_x and A_y, but are labelled with different tile types.

The construction of $q_{\mathfrak{D},c}$ is in several steps, starting with the query $q^i_{\mathfrak{D},c}$ on the left-hand side of Figure 2, where $i \in \{0, \ldots, m-1\}$. In the queries $q^i_{\mathfrak{D},c}$, all the edges represent the role s and v_{ans} is the only answer variable. The edges are undirected because we are working with symmetric roles. Formally,

$$\begin{aligned} q^i_{\mathfrak{D},c} := \{ \; & s(v_{i,0}, v_{i,1}), \ldots, s(v_{i,2m+2}, v_{i,2m+3}), \\ & s(v'_{i,0}, v'_{i,1}), \ldots, s(v'_{i,2m+2}, v'_{i,2m+3}), \\ & s(v_{i,0}, v'_{i,0}), s(v_{i,2m+3}, v'_{i,2m+3}), \\ & s(v, v_{i,0}), s(v, v'_{i,0}), \\ & s(v', v_{i,2m+3}), s(v', v'_{i,2m+3}), \\ & s(v_{\mathsf{ans}}, v_{i,m+1}), s(v_{\mathsf{ans}}, v_{i,m+2}), s(v_{\mathsf{ans}}, v'_{i,m+1}), s(v_{\mathsf{ans}}, v'_{i,m+2}), \\ & G(v), G(v'), A_i(v_{i,0}), \neg A_i(v'_{i,0}), \neg A_i(v_{i,2m+3}), A_i(v'_{i,2m+3}) \; \} \end{aligned}$$

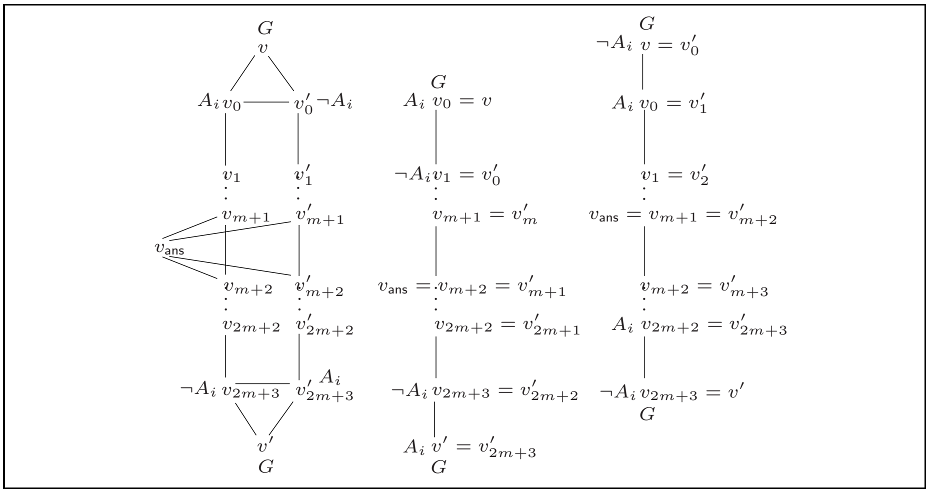


Fig. 2. The query $q^i_{\mathfrak{D},a}$ (left) and two of its collapsings (middle and right)

Observe that we dropped the index "i" to variables in Figure 2. Also observe that all the queries $q^i_{\mathfrak{D},c}$, $i < m$, share the variables v, v', and v_{ans}.

The purpose of the query $q^i_{\mathfrak{D},a}$ is to relate any two G-nodes that agree on the value of the concept name A_i. To explain how this works, we need a few preliminaries. First, a *cycle* in a query is a sequence of distinct nodes $v_0, \dots, v_n$ such that $n \geq 2$, and $s(v_i, v_{i+1}) \in q$ or $s(v_{i+1}, v_i) \in q$ for all $i \leq n$, where $v_{n+1} := v_0$. A query q' is a *collapsing* of a query q if q' is obtained from q by identifying variables. Each match of $q^i_{\mathfrak{D},c}$ in our *tree-structured* grid representation gives rise to a collapsing of $q^i_{\mathfrak{D},c}$ that does not comprise any cycles. To explain how $q^i_{\mathfrak{D},c}$ works, it is helpful to analyze its cycle-free collapsings. We start with the two cycles v, v_0, v'_0 and v', v_{2m+3}, v'_{2m+3}. For eliminating each of these, we have two options:

- to remove the upper cycle, we can identify v with v_0 or v'_0;
- to remove the lower cycle, we can identify v' with v_{2m+3} or v'_{2m+3}.

Observe that if we identify v_0 and v'_0 (or v_{2m+3} and v'_{2m+3}) to collapse the cycle, there will be no matches of the query in any model.

Together, this gives four options for removing the two mentioned length-three cycles. However, two of these options are ruled out because the resulting collapsings have no match in the grid representation. The first such case is when we identify v with v_0 and v' with v_{2m+3}. To see that there is no match, first observe that v_0 and v_{2m+3} have to satisfy G. Then make a case distinction on the two options that we have for eliminating the cycle $\{v_{\mathsf{ans}}, v_{m+1}, v_{m+2}\}$.

Case (1). If we identify v_{ans} and v_{m+1}, the path from the G-variable v_0 to v_{ans} is only of length $m+1$. In our grid representation, all paths from a G-node to an ABox individual (i.e., the root) are of length $m+2$, so there can be no match of this collapsing.

Case (2). If we identify v_{ans} and v_{m+2}, the path from v_{ans} to the G-variable v_{2m+3} is only of length $m+1$ and again there is no match.

We can argue analogously for the case where we identify v with v'_0 and and v' with v'_{2m+3}. Therefore, the two remaining collapsings for eliminating the cycles $\{v, v_0, v'_0\}$ and $\{v', v_{2m+3}, v'_{2m+3}\}$ are the following:

(a) identify v with v_0 and v' with v'_{2m+3};
(b) identify v with v'_0 and v' with v_{2m+3}.

In the first case, we further have to identify v_{ans} with v_{m+2} and v'_{m+1}, for otherwise we can argue as above that there is no match. In the second case, we have to identify v_{ans} with v_{m+1} and v'_{m+2}. After this has been done, there is only one way to eliminate the cycle $v = v_0, \ldots, v_{2m+3}, v' = v'_{2m+3}, \ldots, v'_0$ such that the result is a chain of length $2m+4$ with the G-variables at both ends and the answer variable exactly in the middle (any other way to collapse means that there are no matches). The reflexive loops at the endpoints of the resulting chain and at v_{ans} can simply be dropped since we work with reflexive roles. The resulting cycle-free queries are shown in the middle and right part of Figure 2.

Note that the middle query has A_i at both ends of the chain, and the right one has $\neg A_i$ at the ends. According to our above argumentation, the original query $q^i_{\mathfrak{D},c}$ has a match in the grid representation iff one of these two collapsings has a match. Thus, every match π of $q^i_{\mathfrak{D},c}$ in the grid representation is such that $\pi(v)$ and $\pi(v')$ are (not necessarily distinct) instances of G that agree on the value of A_i.

At this point, a technical remark is in order. Observe that, in the two relevant collapsings of $q^i_{\mathfrak{D},c}$, the end nodes of the chain and their immediate neighbors are labeled dually w.r.t. A_i and $\neg A_i$. This is an artifact of query construction and cannot be avoided. To deal with it, we have introduced F-nodes into our grid representation and ensured that they satisfy Property $\mathsf{T1}$.

Now set $q_{\mathsf{cnt}} := \bigcup_{i<m} q^i_{\mathfrak{D},c}$. It is not hard to see that every match π of q_{cnt} in the grid representation is such that $\pi(v)$ and $\pi(v')$ are (not necessarily distinct) instances of G that have the same A_i-value, for *all* $i < m$. The query q_{cnt} is almost the desired query $q_{\mathfrak{D},c}$. Recall that we want to enforce Condition $(*)$ from above, and thus also need to talk about tile types in the query. The query q_{tile} is given in the left-hand side of Figure 3 for the case of three tiles, i.e., $T = \{0,1,2\}$. In general, for $T = \{0,\ldots,k-1\}$, we define

$$\begin{aligned} q_{\mathsf{tile}} := \bigcup_{i<k} \{ & s(w_{i,0}, w_{i,1}), \ldots, s(w_{i,2m+2}, w_{i,2m+3}), \\ & s(v_{\mathsf{ans}}, w_{i,m+1}), s(v_{\mathsf{ans}}, w_{i,m+2}), \\ & s(v, w_{i,0}), s(v', w_{i,2m+3}), \\ & D_i(w_{i,0}), D_i(w_{i,2m+3})\} \\ \cup & \bigcup_{i<j<k} \{s(w_{i,0}, w_{j,0}), s(w_{i,2m+3}, w_{j,2m+3})\} \\ \cup & \{G(v), G(v')\} \end{aligned}$$

Observe that q_{cnt} and q_{tile} share the variables v, v', and v_{ans}. Also observe that q_{tile} is very similar to the queries $q^i_{\mathfrak{D},c}$, the main difference being the number of

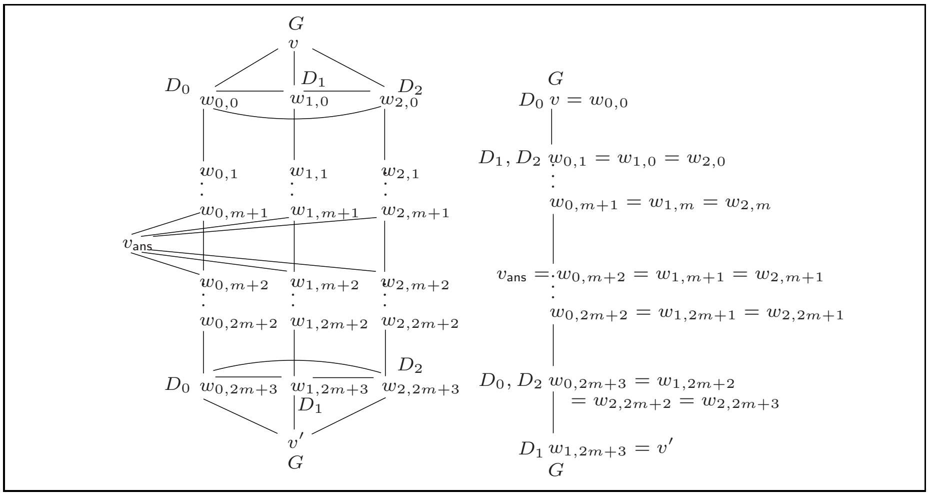


Fig. 3. The query q_{tile} (left) and one of its collapsings (right)

vertical chains. Whereas the queries $q^i_{\mathfrak{D},c}$ have two collapsings that are cycle-free and can have matches in the grid representation, q_{tile} has $k \cdot (k-1)$ such collapsings: for all $i, j \in T$ with $i \neq j$, there is a collapsing into a linear chain of length $2m + 4$ whose two end nodes are labelled D_i and D_j, respectively. An example of such a collapsing is presented on the right-hand side of Figure 3. The arguments for how to obtain these collapsing from q_{tile} and why other collapsings have no match in the grid representation are similar to the line of argumentation used for $q^i_{\mathfrak{D},c}$ and involves Property T2. We refer to [15] for details.

Now, the desired query $q_{\mathfrak{D},c}$ is simply the union of q_{cnt} and q_{tile}. From what was already said about q_{cnt} and q_{tile}, it is easily derived that $q_{\mathfrak{D},c}$ does not match the grid representation iff Property $(*)$ is satisfied. It is possible to show that there is a solution for $\mathfrak{D}$ and c iff $(\emptyset, \mathcal{A}_{\mathfrak{D},c}) \not\models q_{\mathfrak{D},c}$. We have thus proved that rooted query entailment in $\mathcal{ALCI}$ is co-NExpTime-hard. A matching upper bound can be obtained by adapting the techniques in [7]. More details are given in [16].

Theorem 1. *Rooted query entailment in $\mathcal{ALCI}$ is co-*NExpTime*-complete. The lower bound holds even if the TBox is empty and the ABox is of the form $\{C(a)\}$.*

4 2ExpTime-Hardness Results

Theorem 1 shows that, already in the case of rooted queries, conjunctive query entailment in DLs between $\mathcal{ALCI}$ and $\mathcal{SHIQ}$ is more difficult than instance checking. In the general case, conjunctive query entailment in these DLs is even 2ExpTime-complete. The proof is by a reduction of the word problem of exponentially space bounded alternating Turing machines (ATMs) [5], and reuses many ideas from the reduction given in Section 3. Because of space limitations, we can only give a very rough sketch of the proof.

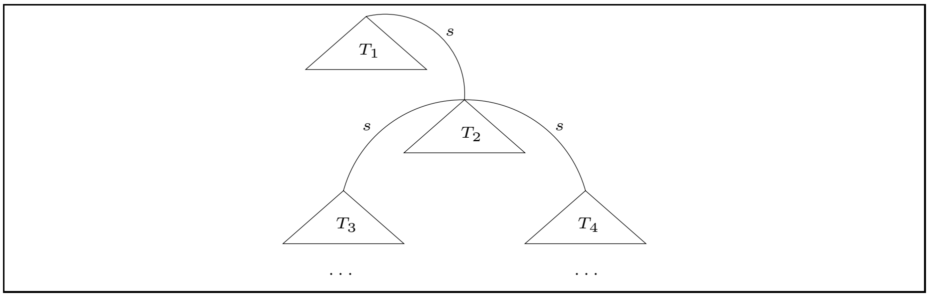


Fig. 4. Representing ATM computations

The main idea is to represent each configuration of an ATM by the leafs of a tree of depth n, similar to the grid representation in Section 3. Trees representing configurations are then interconnected to form a larger tree that represents a computation. This is illustrated in Figure 4. Each of the T_i is a tree of depth n whose leafs represent a configuration. The tree T_1 represents an existential configuration, and thus has only one successor configuration T_2. In contrast, the tree T_2 represents a universal configuration with two successor configurations T_3 and T_4. The difficult part of the reduction is to relate the content of a tape cell in one configuration to the content of the corresponding cell in the successor configurations. The solution is to use queries that are very similar to the query $q_{\mathfrak{D},c}$ employed in the previous section. A few additional technical tricks are needed to achieve directedness (i.e., talking only about successor configurations, but not about predecessor configurations) since we work with symmetric roles. More details of the reduction can be found in [15]. A 2ExpTime upper bound was established in [7] (where also non-simple roles are allowed in the query).

Theorem 2. *Query entailment in $\mathcal{ALCI}$ is 2*ExpTime*-complete. The lower bound holds even for queries without answer variables and for ABoxes of the form $\{C(a)\}$.*

Using Theorem 2, it is also easy to show that admitting transitive roles in the query destroys the better computational properties of rooted query entailment. $\mathcal{ALCI}_{R^+}$ is the extension of $\mathcal{ALCI}$ with transitive roles.

Theorem 3. *Rooted query entailment in $\mathcal{ALCI}_{R^+}$ is 2*ExpTime*-complete if transitive roles are admitted in the query. The lower bound holds even if the TBox contains only transitivity statements and role inclusions, and the ABox is of the form $\{C(a), r(a,a)\}$.*

Proof. (sketch) By Theorem 2, it suffices to establish the lower bound. We reduce non-rooted query entailment in $\mathcal{ALCI}$, which is 2ExpTime-hard by Theorem 2. Let $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ and q be given, with $\mathcal{A} = \{C(a)\}$. Our aim is to construct a knowledge base $\mathcal{K}' = (\mathcal{T}', \mathcal{A}')$ and rooted query q' such that $\mathcal{K} \models q$ iff $\mathcal{K}' \models q'$. Let $C_{\mathcal{T}} = \bigsqcap_{D \sqsubseteq E \in \mathcal{T}} \neg D \sqcup E$. Fix a role name t not occurring in $\mathcal{K}$ and q, and a variable v_0 not occurring in q. Then set

$$\begin{aligned}\mathcal{T}' &:= \{\mathsf{Trans}(t)\} \cup \{r \sqsubseteq t, r^- \sqsubseteq t \mid r \in \mathsf{N_R} \text{ occurs in } \mathcal{K}\}\\ \mathcal{A}' &:= \{C \sqcap \forall t.C_{\mathcal{T}}(a), t(a,a)\}\\ q' &:= q \cup \{t(v_0, v) \mid v \in \mathsf{N_V} \text{ occurs in } q\}.\end{aligned}$$

We make v_0 an answer variable in q'. It is not hard to prove that $\mathcal{T}'$, $\mathcal{A}'$, and q' are as required. ❏

The results proved in this section and the preceeding one show that conjunctive query entailment is computationally hard in fragments of $\mathcal{SHIQ}$ that contain $\mathcal{ALCI}$. In the next section, we prove that inverse roles are indeed the culprit for the high complexity: in $\mathcal{SHQ}$ ($\mathcal{SHIQ}$ without inverse roles), conjunctive query entailment is only ExpTime-complete and thus of the same complexity as instance checking.

5 Query Entailment in $\mathcal{SHQ}$ is ExpTime-Complete

We give an algorithm for query entailment in $\mathcal{SHQ}$ that runs in ExpTime and is inspired by the 2ExpTime algorithm for conjunctive query entailment in $\mathcal{SHIQ}$ given in [7]. The general idea is to (Turing-)reduce query entailment in $\mathcal{SHQ}$ to ABox consistency in $\mathcal{SHQ}^\cap$, i.e., $\mathcal{SHQ}$ extended with role conjunction: given a $\mathcal{SHQ}$-knowledge base $\mathcal{K}$ and a query q, we produce $\mathcal{SHQ}^\cap$-knowledge bases $\mathcal{K}_1, \ldots, \mathcal{K}_n$ such that $\mathcal{K} \not\models q$ iff any of the $\mathcal{K}_i$ is consisent. The construction ensures that n is exponential in the size of $\mathcal{K}$ and q, and the size of each $\mathcal{K}_i$ is polynomial in the size of $\mathcal{K}$ and q. Since knowledge base consistency in $\mathcal{SHQ}^\cap$ can be decided in ExpTime, we obtain the desired ExpTime upper bound for query entailment in $\mathcal{SHQ}$. Proof details for the lemmas presented in this section can be found in [16].

We start with proving an $\mathcal{SHQ}$ counterpart of Lemma 1. Let $\mathcal{J}$ be an interpretation. A *forest base* $\mathcal{J}$ is an interpretation that interprets transitive roles in an arbitrary way (i.e., not necessarily transitively) and where (i) $\Delta^{\mathcal{J}}$ is a prefix-closed subset of $\mathbb{N}^+$ and (ii) if $(d,e) \in r^{\mathcal{J}}$, then $e, d \in \mathbb{N}$ or $e = d \cdot c$ for some $c \in \mathbb{N}$. Elements of $\Delta^{\mathcal{J}} \cap \mathbb{N}$ are called the *roots* of $\mathcal{J}$. An interpretation $\mathcal{I}$ is the $\mathcal{K}$*-closure* of $\mathcal{J}$ if $\mathcal{I}$ is identical to $\mathcal{J}$ except that, for all roles r, we have

$$r^{\mathcal{I}} = r^{\mathcal{J}} \cup \bigcup_{\mathcal{K} \models s \sqsubseteq r \wedge \mathcal{K} \models \mathsf{Trans}(s)} (s^{\mathcal{J}})^+.$$

A model $\mathcal{I}$ of a knowledge base $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ is a *forest model* of $\mathcal{K}$ if (iii) $\mathcal{I}$ is the $\mathcal{K}$-closure of a forest base $\mathcal{J}$, and (iv) for every root d of $\mathcal{J}$, there is an $a \in \mathsf{Ind}(\mathcal{A})$ such that $a^{\mathcal{I}} = d$. The *roots* of $\mathcal{I}$ are defined as the roots of $\mathcal{J}$. The following proposition shows that, when deciding conjunctive query entailment in $\mathcal{SHQ}$, it suffices to concentrate on forest models.

Proposition 1. *Let $\mathcal{K}$ be an $\mathcal{SHQ}$-knowledge base and q a conjunctive query. If $\mathcal{K} \not\models q$, then there is a forest model $\mathcal{I}$ of $\mathcal{K}$ such that $\mathcal{I} \not\models q$.*

Throughout this section, we will sometimes view a conjunctive query as a directed graph $G_q = (V_q, E_q)$ with $V_q = \mathsf{Var}(q)$ and $E_q = \{(v, v') \mid r(v, v') \in q$ for some $r \in \mathsf{N_R}\}$. We call q *tree-shaped* if G_q is a tree. If q is tree-shaped and v is the root of G_q, we call v the root of q.

In the following, we introduce three notions that are central to the construction of the knowledge bases $\mathcal{K}_1, \ldots, \mathcal{K}_n$: fork rewritings, splittings, and spoilers. We start with fork rewritings, and say that

– q' is *obtained from q by fork elimination* if q' is obtained from q by selecting two atoms $r(v', v)$ and $s(v'', v)$ with $v' \neq v''$ and identifying v' and v'';
– q' is a *fork rewriting* of q if q' is obtained from q by repeated (but not necessarily exhaustive) fork elimination;
– q' is a *maximal fork rewriting* of q if q' is a fork rewriting and no further fork elimination is possible in q'.

The following lemma allows us to speak of *the* maximal fork rewriting of a conjunctive query.

Lemma 2. *Modulo variable renaming, every conjunctive query has a unique maximal fork rewriting.*

Now for splittings, which are partitions of the variables in (a fork rewriting of) the input query. Intuitively, a splitting is induced by each match π for some forest model $\mathcal{I}$ of the input KB $\mathcal{K}$ and the input query q. More precisely, each variable $v \in \mathsf{Var}(q)$ is either

(a) mapped to a root $\pi(v)$ of $\mathcal{I}$;
(b) mapped to a non-root $\pi(v)$ of $\mathcal{I}$ such that there is a variable v' mapped to a root $\pi(v')$ of $\mathcal{I}$ and with v reachable from v' in G_q;
(c) mapped to a non-root $\pi(v)$ of $\mathcal{I}$, but does not satisfy Condition (b).

The purpose of splittings is to describe such a partition without reference to a concrete model $\mathcal{I}$ and a concrete match π. Let $\mathcal{K}$ be an $\mathcal{SHQ}$-knowledge base. A *splitting* of q w.r.t. $\mathcal{K}$ is a tuple $\Pi = \langle R, T, S_1, \ldots, S_n, \mu, \nu \rangle$, where $R, T, S_1, \ldots, S_n$ is a partitioning of $\mathsf{Var}(q)$, $\mu : \{1, \ldots, n\} \to R$ assigns to each set S_i a variable $\mu(i)$ in R, and $\nu : R \to \mathsf{Ind}(\mathcal{A})$ assigns to each variable in R an individual in $\mathcal{A}$. A splitting has to satisfy the following conditions, where $q|_V$ denotes the restriction of q to $V \subseteq \mathsf{Var}(q)$:

1. the query $q|_T$ is a variable-disjoint union of tree-shaped queries;
2. the queries $q|_{S_i}$, $1 \leq i \leq n$, are tree-shaped;
3. if $r(v, v') \in q$, then one of the following holds: (i) v, v' belong to the same set $R, T, S_1, \ldots, S_n$ or (ii) $v \in R$, $\mu(i) = v$, and $v' \in S_i$ is the root of $q|_{S_i}$;
4. for $1 \leq i \leq n$, there is an atom $r(\mu(i), v_0) \in q$, with v_0 the root of $q|_{S_i}$.

Intuitively, the R component of a splitting corresponds to Case (a) above, the $S_1, \ldots, S_n$ correspond to Case (b), and T corresponds to Case (c). Before we introduce spoilers, we establish a central lemma about splittings. We start with a preliminary. Let q be a tree-shaped conjunctive query. We define a $\mathcal{SHQ}^\sqcap$-concept $C_{q,v}$ for each variable $v \in \mathsf{Var}(q)$:

– if v is a leaf in G_q, then $C_{q,v} = \bigsqcap_{C(v) \in q} C$;
– otherwise, $C_{q,v} = \bigsqcap_{C(v) \in q} C \sqcap \bigsqcap_{(v,v') \in E_q} \exists (\bigcap_{s(v,v') \in q} s).C_{q,v'})$.

If v is the root of q, we use C_q to abbreviate $C_{q,v}$. Observe that, since we allow only simple roles in a query q, all concepts C_q involve only simple roles inside role conjunction. The following lemma establishes a connection between forest models and splittings of fork rewritings.

Lemma 3. *Let $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ be a knowledge base, $\mathcal{I}$ a forest model of $\mathcal{K}$, and q a conjunctive query. Then $\mathcal{I} \models q$ iff there exists a fork rewriting q' of q and a splitting $\langle R, T, S_1, \ldots, S_n, \mu, \nu \rangle$ of q' w.r.t. $\mathcal{K}$ such that*

1. *for each disconnected component $\widehat{q}$ of T, there is a $d \in \Delta^{\mathcal{I}}$ with $d \in (C_{\widehat{q}})^{\mathcal{I}}$;*
2. *if $C(v) \in q'$ with $v \in R$, then $\nu(v)^{\mathcal{I}} \in C^{\mathcal{I}}$;*
3. *if $r(v, v') \in q'$ with $v, v' \in R$, then $(\nu(v)^{\mathcal{I}}, \nu(v')^{\mathcal{I}}) \in r^{\mathcal{I}}$;*
4. *for $1 \leq i \leq n$, we have $\nu(\mu(i))^{\mathcal{I}} \in \left(\exists (\bigcap_{s(\mu(i),v_0) \in q'} s).C_{q'|_{S_i}}\right)^{\mathcal{I}}$ with v_0 root of the tree-shaped query $q'|_{S_i}$.*

Now for the definition of spoilers, which exploit Lemma 3 to prevent matches of the input query q in forest-models of the input KB $\mathcal{K} = (\mathcal{T}, \mathcal{A})$. We first define spoilers of specific splittings, and then spoilers of the query (i.e., of all splittings). Let $\Pi = \langle R, T, S_1, \ldots, S_n, \mu, \nu \rangle$ be a splitting of q w.r.t. $\mathcal{K}$ such that $q_1, \ldots, q_k$ are the (tree-shaped) disconnected components of $q|_T$. A $\mathcal{SHQ}^{\cap}$-knowledge base $(\mathcal{T}', \mathcal{A}')$ is a *spoiler* for q, $\mathcal{K}$, and Π if one of the following conditions hold:

1. $\top \sqsubseteq \neg C_{q_i} \in \mathcal{T}'$, for some i with $1 \leq i \leq k$;
2. there is an atom $C(v) \in q$ with $v \in R$ and $\neg C(\nu(v)) \in \mathcal{A}'$;
3. there is an atom $r(v, v') \in q$ with $v, v' \in R$ and $\neg r(\nu(v), \nu(v')) \in \mathcal{A}'$;
4. $\neg D(\nu(\mu(i))) \in \mathcal{A}'$ for some $i \in \{1, \ldots, n\}$, and where $D = \exists (\bigcap_{s(\mu(i),v_0) \in q} s).C_{q|_{S_i}}$ with v_0 root of $q|_{S_i}$.

A $\mathcal{SHQ}^{\cap}$-knowledge base $\mathcal{K}' = (\mathcal{T}', \mathcal{A}')$ is a *spoiler* for q and $\mathcal{K}$ if (i) for every fork rewriting q' of q and every splitting Π of q' w.r.t. $\mathcal{K}$, $\mathcal{K}'$ is a spoiler for q', $\mathcal{K}$, and Π; and (ii) $\mathcal{K}'$ is minimal with Property (i). The proof of the following lemma is based on the correspondence between Conditions 1-4 of spoilers and Conditions 1-4 of Lemma 3.

Lemma 4. *Let $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ be a $\mathcal{SHQ}$-knowledge base and q a conjunctive query. Then $\mathcal{K} \not\models q$ iff there is a spoiler $(\mathcal{T}', \mathcal{A}')$ for q and $\mathcal{K}$ such that $(\mathcal{T} \cup \mathcal{T}', \mathcal{A} \cup \mathcal{A}')$ is consistent.*

Lemma 4 suggests the following algorithm for deciding conjunctive query entailment in $\mathcal{SHQ}$: given $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ and q, enumerate all spoilers $(\mathcal{T}', \mathcal{A}')$ for q and $\mathcal{K}$, return "yes" if for all such spoilers, $(\mathcal{T} \cup \mathcal{T}', \mathcal{A} \cup \mathcal{A}')$ is inconsistent, and "no" otherwise. To prove that this algorithm runs in ExpTime, we first note that consistency of $\mathcal{SHQ}^{\cap}$-KBs is ExpTime-complete. Since only simple roles occur inside role conjunctions, this can be proved by an easy variation of Lemma 6.19 in [22]. It thus suffices to establish the following.

Lemma 5. *Let $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ be a $\mathcal{SHQ}$-knowledge base and q a conjunctive query. Then the number of spoilers for q and $\mathcal{K}$ is exponential in the size of q and $\mathcal{K}$ and the set of all spoilers can be computed in time exponential in the size of q and $\mathcal{K}$.*

The proof of this lemma is a key ingredient to our ExpTime upper bound. The upper bound on the number of spoilers is established by showing that (i) all individual names and role names occurring in spoilers also occur in the input KB and input query, and (ii) there are only polynomially many different concepts that can occur in spoilers. While (i) is trivial, (ii) is not. Define

$$\mathsf{Trees}(q) := \{q|_{\mathsf{Reach}_q(v)} \mid v \in \mathsf{Var}(q) \text{ and } q|_{\mathsf{Reach}_q(v)} \text{ is tree-shaped}\}.$$

The proof of (ii) proceeds by showing that if C occurs in a spoiler of $\mathcal{K}$ and q and q^* is the maximal fork rewriting of q, then there is a $\widehat{q} \in \mathsf{Trees}(q)$ with $C = C_q$. Details are given in [16].

Summing up, we have established the following result, where the lower bound is trivial by a reduction of instance checking in $\mathcal{SHQ}$.

Theorem 4. *Conjunctive query entailment in $\mathcal{SHQ}$ is* ExpTime*-complete.*

6 Conclusion

We have carried out a detailed investigation of the complexity of conjunctive query entailment in DLs between $\mathcal{ALC}$ and $\mathcal{SHIQ}$. In particular, we have proved that conjunctive query entailment is more complex than instance checking when inverse roles are present (2ExpTime vs ExpTime), and that the complexity coincides without inverse roles (ExpTime). Our two upper bound proofs (Theorem 1 and 4) do not apply to the case where transitive roles are admitted in the query. As shown by Theorem 3, the NExpTime upper bound from Theorem 1 cannot be generalized to this case. It remains an open problem whether or not the ExpTime upper bound in Theorem 4 can be adapted to $\mathcal{SHQ}$ with transitive roles in the query. An ExpTime upper bound for a fragment of this problem is established in [19].

References

1. Baader, F., McGuiness, D.L., Nardi, D., Patel-Schneider, P.: The Description Logic Handbook. Cambridge University Press, Cambridge (2003)
2. Calvanese, D., De Giacomo, G., Lenzerini, M.: On the decidability of query containment under constraints. In: Proc. of PODS 1998, pp. 149–158 (1998)
3. Calvanese, D., Giacomo, G.D., Lembo, D., Lenzerini, M., Rosati, R.: Data complexity of query answering in description logics. In: Proc. of KR 2006, pp. 260–270. AAAI Press, Menlo Park (2006)
4. Calvanese, D., Eiter, T., Ortiz, M.: Answering regular path queries in expressive description logics: an automata-theoretic approach. In: Proc. of AAAI 2007. AAAI Press, Menlo Park (2007)

5. Chandra, A.K., Kozen, D.C., Stockmeyer, L.J.: Alternation. Journal of the ACM 28(1), 114–133 (1981)
6. Glimm, B., Horrocks, I., Sattler, U.: Conjunctive query entailment for $\mathcal{SHOQ}$. In: Proc. of DL 2007. CEUR-WS, vol. 250 (2007)
7. Glimm, B., Lutz, C., Horrocks, I., Sattler, U.: Answering conjunctive queries in the $\mathcal{SHIQ}$ description logic. JAIR 31, 150–197 (2008)
8. Horrocks, I., Sattler, U., Tobies, S.: Practical reasoning for expressive description logics. In: Ganzinger, H., McAllester, D., Voronkov, A. (eds.) LPAR 1999. LNCS, vol. 1705, pp. 161–180. Springer, Heidelberg (1999)
9. Horrocks, I., Sattler, U., Tobies, S.: Reasoning with individuals for the description logic SHIQ. In: McAllester, D. (ed.) CADE 2000. LNCS, vol. 1831, pp. 482–496. Springer, Heidelberg (2000)
10. Horrocks, I., Tessaris, S.: A conjunctive query language for description logic ABoxes. In: Proc. of AAAI 2000. AAAI Press, Menlo Park (2000)
11. Hustadt, U., Motik, B., Sattler, U.: Data complexity of reasoning in very expressive description logics. In: Proc. of IJCAI 2005, pp. 466–471. Professional Book Center (2005)
12. Krötzsch, M., Rudolph, S., Hitzler, P.: Conjunctive queries for a tractable fragment of OWL 1.1. In: Aberer, K., Choi, K.-S., Noy, N., Allemang, D., Lee, K.-I., Nixon, L., Golbeck, J., Mika, P., Maynard, D., Mizoguchi, R., Schreiber, G., Cudré-Mauroux, P. (eds.) ISWC 2007. LNCS, vol. 4825, pp. 310–323. Springer, Heidelberg (2007)
13. Lutz, C.: The Complexity of Reasoning with Concrete Domains. PhD thesis, LuFG Theoretical Computer Science, RWTH Aachen, Germany (2002)
14. Lutz, C., Areces, C., Horrocks, I., Sattler, U.: Keys, nominals, and concrete domains. Journal of Artificial Intelligence Research (JAIR) 23, 667–726 (2005)
15. Lutz, C.: Inverse roles make conjunctive queries hard. In: Proc. of DL 2007, CEUR-WS vol. 250 (2007), http://lat.inf.tu-dresden.de/~clu/papers/
16. Lutz, C.: Two upper bounds for conjunctive query answering in $\mathcal{SHIQ}$. In: Proc. of DL 2008, CEUR-WS (2008), http://lat.inf.tu-dresden.de/~clu/papers/
17. Ortiz, M., Calvanese, D., Eiter, T.: Characterizing data complexity for conjunctive query answering in expressive description logics. In: Proc. of AAAI 2006. AAAI Press, Menlo Park (2006)
18. Ortiz, M., Šimkus, M., Eiter, T.: Worst-case optimal conjunctive query answering for an expressive description logic without inverses. In: Proc. of AAAI 2008. AAAI Press, Menlo Park (2008)
19. Ortiz, M., Šimkus, M., Eiter, T.: Conjunctive query answering in $\mathcal{SH}$ using knots. In: Proc. of DL 2008. CEUR WS (2008)
20. Schaerf, A.: On the complexity of the instance checking problem in concept languages with existential quantification. JIIS 2, 265–278 (1993)
21. Rosati, R.: On conjunctive query answering in $\mathcal{EL}$. In: Proc. of DL 2007. CEUR-WS, vol. 250 (2007)
22. Tobies, S.: Complexity Results and Practical Algorithms for Logics in Knowledge Representation. PhD thesis, RWTH Aachen (2001)

A General Tableau Method for Deciding Description Logics, Modal Logics and Related First-Order Fragments

Renate A. Schmidt and Dmitry Tishkovsky

School of Computer Science, The University of Manchester
{renate.schmidt,dmitry.tishkovsky}@manchester.ac.uk

Abstract. This paper presents a general method for proving termination of tableaux-based procedures for modal-type logics and related first-order fragments. The method is based on connections between filtration arguments and tableau blocking techniques. The method provides a general framework for developing tableau-based decision procedures for a large class of logics. In particular, the method can be applied to many well-known description and modal logics. The class includes traditional modal logics such as S4 and modal logics with the universal modality, as well as description logics such as $\mathcal{ALC}$ with nominals and general TBoxes. Also contained in the class are harder and less well-studied modal logics with complex modalities and description logics with complex role operators such as Boolean modal logic, and the description logic $\mathcal{ALBO}$. In addition, the techniques allow us to specify tableau-based decision procedures for related solvable fragments of first-order logic, including the two-variable fragment of first-order logic.

1 Introduction

This research is motivated by the current absence of general decision procedures in automated reasoning for tableaux and instantiation-based methods. Although tableau and instantiation-based methods are popular and various general theorem provers exist, so far not much research has been undertaken in using these approaches as decision procedures for solvable first-order classes. This contrasts with the situation of resolution, where a good understanding exists of turning resolution procedures into decision procedures and most of the standard solvable classes of first-order logic have been shown to be decidable by resolution. Tableau methods are however successfully been used for developing decision procedures in the areas of description logic and modal logic. A crucial ingredient of tableau decision procedures for description and modal logics are blocking techniques, which are needed to detect repetitions in the generated models. For different logics various different blocking techniques have been developed and implemented (e.g. [2,12]).

The second motivation for this research is to provide theoretical foundations for generic platforms in which tableau decision procedures can be built in a uniform way for different logics and different applications. Examples of existing tableau prover engineering platforms are the Logics Workbench [11], LoTREC [9], and the Tableaux Work Bench [1]. These provide flexible and general frameworks for defining and experimenting with implementations of different sets of tableau inference rules for different logics, different rule application strategies and different optimisations. These systems provide generic provers for standard modal logics, temporal logics and propositional dynamic logic. As yet the infrastructure is not general enough to support the building of tableau decision procedures for description logics and dynamic modal logics with complex relational constructs, which do not necessarily have the tree model property. For these kinds of logics the standard blocking techniques are not sufficient [14].

In [14] we introduced a general blocking mechanism, called *unrestricted blocking*, which is suitable to decide the logics of interest. The blocking mechanism also has the appropriate flexibility and uniformity required for generic prover engineering platforms, because it can be used to simulate existing standard blocking mechanisms such a subset blocking, equality blocking, dynamic blocking, etc.

In this paper we present a general method for proving termination of tableau-based procedures for modal-type logics and related first-order fragments. The central observation underlying the method is that most termination proofs for tableau decision procedures (regardless of which form of blocking techniques they employ), can be seen to exploit techniques which look similar to filtration arguments, standard in logic and algebra for proving finite model property results. Our method provides a framework to turn given sound and complete tableau calculi into decision procedures by enhancing them with the unrestricted blocking mechanism.

The method can be applied to many description and modal logics, as well as first-order fragments. For illustration purposes we discuss in detail how the method can be applied to two description logics. In particular, we apply it to the description logic $\mathcal{ALCO}$ with transitive roles, referred to as $\mathcal{SO}$. $\mathcal{SO}$ corresponds to the (multi-modal version) of the modal logic $\mathsf{S4}$ with nominals ($\mathsf{S4}$ is the modal logic of pre-orders). We also apply the method to the description logic $\mathcal{ALBO}$, which is an extension of $\mathcal{ALCO}$ with the Boolean operators on roles and role inverse. We show how the method can be used to obtain two tableau decision procedures for $\mathcal{ALBO}$. One is equivalent to the tableau decision procedure first developed in [14] and constructs standard models. The second tableau decision procedure has fewer rules and constructs quasi-models.

The paper is structured as follows. Section 2 describes the general framework of the method and identifies general properties sufficient to turn existing tableau calculi into decision procedures. This also gives an abstract formalisation of a class of logics decidable by tableau methods. The subsequent sections are devoted to showing in detail how the method can be used. Section 3 gives some preliminary definitions. In Section 4 we then show how the method can be applied to the description logic $\mathcal{SO}$. Section 5 discusses the case of the description logic $\mathcal{ALBO}$ in some detail. Other applications, consequences and possibilities are discussed in Section 6.

Due to lack of space many proofs are omitted, but can be found in the long version [13] of the paper. We assume the reader has basic familiarity with description logic and modal logic.

2 General Framework

This section describes the general method of our framework. Let L denote an abstract description logic. The aim is to formulate general conditions under which terminating tableau procedures exist for L.

Syntax. In this paper it is assumed that L is an extension of the description logic $\mathcal{ALCO}$ with role operators $\rho_0, \rho_1, \ldots$. Formally, L is defined over a signature which is a triple $(\mathsf{O}, \mathsf{C}, \mathsf{R})$ of three disjoint alphabets: the alphabet of symbols for individuals (or objects) $\mathsf{O} = \{\ell_0, \ell_1, \ldots\}$, the alphabet of concept symbols $\mathsf{C} = \{p_0, p_1, \ldots\}$, and the alphabet of role symbols $\mathsf{R} = \{r_0, r_1, \ldots\}$. The logical operators are: $\neg$ (*negation*), $\sqcup$ (*union*), $\exists$ (*existential concept restriction*), and the role operators $\rho_0, \rho_1, \ldots$. The role operators are assumed to have finite arities $\mu_0, \mu_1, \ldots$, respectively. *Concept expressions* (or *concepts*) and *role expressions* (or *roles*) are defined as follows:

$$\begin{aligned} C, D &\overset{\mathsf{def}}{=} p \mid \{\ell\} \mid \neg C \mid C \sqcup D \mid \exists R.C, \\ R, R_i &\overset{\mathsf{def}}{=} r \mid \rho_0(R_1, \ldots, R_{\mu_0}) \mid \rho_1(R_1, \ldots, R_{\mu_1}) \mid \ldots, \end{aligned}$$

where p denotes an atomic concept in C, ℓ denotes an individual in O, and r denotes an atomic role in R.

Concepts of the form $\{\ell\}$ are called *singleton concepts*, or *nominals*. Concept expressions, role expressions and individuals are collectively referred to as L-*expressions.*

Semantics. Let $C_0, C_1, \ldots$ be an enumeration of all concepts and $R_0, R_1, \ldots$ an enumeration of all roles in the signature of L. An *L-model* $\mathcal{I}$ (or an *L-interpretation*) of an L-signature Σ is a tuple $(\Delta^{\mathcal{I}}, C_0^{\mathcal{I}}, \ldots, \ell_0^{\mathcal{I}}, \ldots, R_0^{\mathcal{I}}, \ldots)$, where $\Delta^{\mathcal{I}}$ is a non-empty set, each $C_i^{\mathcal{I}}$ is a subset of $\Delta^{\mathcal{I}}$, $\ell_i^{\mathcal{I}} \in \Delta^{\mathcal{I}}$, and each $R_0^{\mathcal{I}}$ is a binary relation over $\Delta^{\mathcal{I}}$. Further, for any expressions of Σ the following conditions must be satisfied:

$$\{\ell\}^{\mathcal{I}} = \{\ell^{\mathcal{I}}\}, \qquad (\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}, \qquad (C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}},$$
$$(\exists R.C)^{\mathcal{I}} = \{x \mid \exists y \in C^{\mathcal{I}}\ (x, y) \in R^{\mathcal{I}}\}, \text{ and}$$

additional semantic conditions for each of the role operators $\rho_0, \rho_1, \ldots$ of L.

We say $\mathcal{I}$ is a *(full) L-model* if it is an L-model of the signature consisting of all L-expressions.

Closure operator. Let sub be an *(expression) closure* operator over sets of L-expressions, i.e. for every set S of L-expressions, $\mathsf{sub}(S)$ is a set of L-expressions containing the set S. We define $\mathsf{sub}(E_1, \ldots, E_n) \overset{\text{def}}{=} \mathsf{sub}(\{E_1, \ldots, E_n\})$ for any finite sequence of expressions $E_1, \ldots, E_n$. We say sub is *finite* if $\mathsf{sub}(E_1, \ldots, E_n)$ is finite for any expressions $E_1, \ldots, E_n$. An L*-signature* is a set Σ of L-expressions closed under sub. The closure operator sub is *adequate* to the semantics of L if every L-model of an arbitrary signature Σ can be transformed to a full L-model preserving interpretations of any expressions from the signature. In the sequel, we always assume that sub is adequate to the semantics of the considered logic.

Finite filtration. Let $\sim$ be an equivalence relation on an L-model $\mathcal{I}$ and let $[x] \overset{\text{def}}{=} \{y \in \Delta^{\mathcal{I}} \mid x \sim y\}$. Let $\overline{\mathcal{I}} = (\Delta^{\overline{\mathcal{I}}}, C_0^{\overline{\mathcal{I}}}, \ldots, \ell_0^{\overline{\mathcal{I}}}, \ldots, R_0^{\overline{\mathcal{I}}}, \ldots)$ be a structure which satisfies the following conditions: $\Delta^{\overline{\mathcal{I}}} = \{[x] \mid x \in \Delta^{\mathcal{I}}\}$, $C^{\overline{\mathcal{I}}} = \{[x] \mid x \in C^{\mathcal{I}}\}$ for every concept C, $\ell^{\overline{\mathcal{I}}} = [\ell^{\mathcal{I}}]$ for every $\ell \in \mathsf{O}$, and $([x], [y]) \in R^{\overline{\mathcal{I}}}$ whenever $\exists x' \sim x \exists y' \sim y\, (x', y') \in R^{\mathcal{I}}$ for every role R. A structure $\overline{\mathcal{I}}$ for which these conditions hold is called a $\sim$*-filtration* of $\mathcal{I}$.

We say that L *admits finite filtration* iff for every finite L-signature Σ and every L-model $\mathcal{I}$ of the signature Σ there exists an equivalence relation $\sim$ on $\mathcal{I}$ such that there is a $\sim$-filtration $\overline{\mathcal{I}}$ of $\mathcal{I}$ which is a finite L-model of the signature Σ.

Theorem 1. *Let L be a logic and sub be a finite expression closure operator. If L admits finite filtration then L has the finite model property.*

Tableau calculus. It is convenient to let the tableau calculus operate on labelled concept expressions, which are statements of the form $\ell : C$, where ℓ denotes an individual and C denotes a concept expression. Our language of tableau calculi is therefore a suitable extension of the language of L. So is the semantics. In particular, the semantics of labelled concept expressions is specified by:

$$(\ell : C)^{\mathcal{I}} = \begin{cases} \Delta^{\mathcal{I}}, & \text{if } \ell^{\mathcal{I}} \in C^{\mathcal{I}}, \\ \emptyset, & \text{otherwise,} \end{cases}$$

where $\mathcal{I}$ denotes any L-model.

In general, a *tableau inference rule*[1] is a relation

$$\frac{\ell_1 : C_1, \quad \ldots, \quad \ell_n : C_n}{\ell_1^1 : D_1^1, \quad \ldots, \quad \ell_{k_1}^1 : D_{k_1}^1 \quad \mid \cdots \mid \quad \ell_1^m : D_1^m, \quad \ldots, \quad \ell_{k_m}^m : D_{k_m}^m},$$

where $n \geq 1$, $m \geq 0$, each $k_i \geq 1$, and all atomic concepts and atomic roles occurring below the (horizontal) line in the concepts, also occur in some concept(s) above the line. If an individual ℓ occurs in the statements below the line but not in statements above it then ℓ is assumed to be a newly chosen individual. In this case the rule is said to be *individual generating* and we usually specify the new individuals to be generated in the side conditions on the right of a rule. A

[1] As implemented in MetTeL system [15]

$$(\bot)\colon \frac{\ell : C,\quad \ell : \neg C}{\bot} \qquad (\neg\neg)\colon \frac{\ell : \neg\neg C}{\ell : C}$$

$$(\neg\sqcup)\colon \frac{\ell : \neg(C \sqcup D)}{\ell : \neg C,\quad \ell : \neg D} \qquad (\sqcup)\colon \frac{\ell : (C \sqcup D)}{\ell : C \mid \ell : D}$$

$$(\mathsf{sym})\colon \frac{\ell : \{\ell'\}}{\ell' : \{\ell\}} \qquad (\neg\mathsf{sym})\colon \frac{\ell : \neg\{\ell'\}}{\ell' : \neg\{\ell\}} \qquad (\mathsf{mon})\colon \frac{\ell : \{\ell'\},\quad \ell' : C}{\ell : C} \qquad (\mathsf{id})\colon \frac{\ell : C}{\ell : \{\ell\}}$$

$$(\exists)\colon \frac{\ell : \exists R.C}{\ell : \exists R.\{\ell'\},\quad \ell' : C}(\ell' \text{ is new}) \qquad (\neg\exists)\colon \frac{\ell : \neg\exists R.C,\quad \ell : \exists R.\{\ell'\}}{\ell' : \neg C}$$

Fig. 1. Tableau calculus $T_{\mathcal{ALCO}}$ for $\mathcal{ALCO}$

tableau closure rule is a tableau inference rule with no conclusions, i.e. $m = 0$. In this case we use the standard notation

$$\frac{\ell_1 : C_1,\quad \ldots,\quad \ell_n : C_n}{\bot}$$

to distinguish it from other tableau rules.

A *tableau calculus* is a set of tableau inference rules. Inference steps in a tableau calculus are performed in the usual way. A rule is applied to a set of statements in a branch of a tableau, if the statements are instances of the premises of the rule. Then, in the case of a non-branching rule, the corresponding instances of the conclusions of the rule are added to the branch. A branching rule splits the branch into several branches and adds the corresponding instances of the conclusions to each branch.

Let T denote a tableau calculus and C a concept. We take an arbitrary individual ℓ which does not occur in C. We denote by $T(C)$ a finished tableau built using the rules of T, starting with the concept $\ell : C$ as input. That is, we assume that all branches in the tableau are expanded and all applicable rules of T have been applied in $T(C)$. As usual we assume that all the rules of the calculus are applied *non-deterministically to a tableau.* A branch of a tableau is *closed* if a contradiction has been derived in this branch via tableau closure rules, otherwise the branch is called *open.* The tableau $T(C)$ is *closed* if all its branches are closed and $T(C)$ is *open* otherwise. We say that T is *terminating* iff for every concept C either $T(C)$ is finite whenever $T(C)$ is closed or $T(C)$ has a finite open branch if $T(C)$ is open. T is *sound* iff C is unsatisfiable whenever $T(C)$ is closed for all concepts C. T is *complete* iff for any concept C, C is satisfiable (has a model) whenever $T(C)$ is open.

Let T_L be a tableau calculus for L. We assume that the calculus includes at least the rules for $\mathcal{ALCO}$ given in Figure 1. T_L includes in addition tableau rules to handle the role operators $\rho_0, \rho_1, \ldots$ of L. For simplicity, in this paper we assume that ($\exists$) is the only individual generating rule in T_L. We also require that, in T_L, if an individual ℓ occurs in some expression in a branch then $\ell : \{\ell\}$ also appears in the branch. (The (id) rule ensures this in $T_{\mathcal{ALCO}}$. It can be proved that $T_{\mathcal{ALCO}}$ is sound, complete, and terminating.)

For every open branch $\mathcal{B}$ in a T_L-tableau we define the following equivalence relation $\sim_\mathcal{B}$: $\ell \sim_\mathcal{B} \ell' \overset{\mathsf{def}}{\iff} \ell : \{\ell'\} \in \mathcal{B}$ for any individuals ℓ and ℓ' in the branch $\mathcal{B}$. That $\sim_\mathcal{B}$ is indeed an equivalence relation follows from the presence of the rules (id), (sym), and (mon). Let $\|\ell\| \overset{\mathsf{def}}{=} \{\ell' \mid \ell \sim_\mathcal{B} \ell'\}$.

A tableau T_L is called *constructively complete* for L iff (i) it is complete and (ii) for any satisfiable concept C and any open branch $\mathcal{B}$ in $T_L(C)$ there is an L-model $\mathcal{I}(\mathcal{B})$ satisfying C such that $\Delta^{\mathcal{I}(\mathcal{B})} = \{\|\ell\| \mid \ell : \{\ell\} \in \mathcal{B}\}$,

– $\ell : D \in \mathcal{B}$ implies $\|\ell\| \in D^{\mathcal{I}(\mathcal{B})}$, and
– $\ell : \exists R.\{\ell'\} \in \mathcal{B}$ implies $(\|\ell\|, \|\ell'\|) \in R^{\mathcal{I}(\mathcal{B})}$.

We say T_L is *compatible with* sub iff for any concept C if a statement $\ell : D$ appears in the tableau $T_L(C)$ then either $D \in \mathsf{sub}(C)$, $D = \{\ell'\}$, $D = \neg\{\ell'\}$, $D = \exists R.\{\ell'\}$, or $D = \neg\exists R.\{\ell'\}$, for some role R from $\mathsf{sub}(C)$ and an individual ℓ'.

Blocking. We add the *unrestricted blocking* rule (ub) from [14] to the calculus:

$$\text{(ub)}: \frac{\ell : \{\ell\}, \quad \ell' : \{\ell'\}}{\ell : \{\ell'\} \mid \ell : \neg\{\ell'\}}.$$

The same conditions as in [14] are required to hold. The conditions use an ordering on individuals which reflects the order in which the individuals are introduced during the derivation by the individual generating rules. Specifically, let $<$ be an ordering on individuals in the branch which is a linear extension of the order of introduction of the individuals during the derivation in T_L, i.e. let $\ell < \ell'$, whenever the first appearance of individual ℓ' in the branch is strictly later than the first appearance of individual ℓ.

The conditions that blocking must satisfy are:

(c1) As usual any inference rule is applied at most once to the same set of premises.
(c2) The ($\exists$) rule is applied only to expressions of the form $\ell : \exists R.C$, when C is not a singleton, i.e. $C \neq \{\ell''\}$ for some individual ℓ''.
(c3) If $\ell : \{\ell'\}$ appears in a branch and $\ell < \ell'$ then all further applications of the ($\exists$) rule to expressions of the form $\ell' : \exists R.C$ are not performed within the branch.
(c4) In every open branch there is some node from which point onwards before any application of the ($\exists$) rule all possible applications of the (ub) rule have been performed.

General termination. By T_L + (ub) we denote the calculus comprising of the rules of T_L (including those of $T_{\mathcal{ALCO}}$) and the unrestricted blocking rule (ub). It is further required that the calculus satisfies the conditions (c1)–(c4), above.

The main result of the paper is following.

Theorem 2. *Let L be an extension of the description logic $\mathcal{ALCO}$. T_L + (ub) is sound, complete, and terminating tableau calculus for L, if the following conditions all hold:*

1. sub *is a finite closure operator for L-expressions.*
2. *L is a logic which admits finite filtration.*
3. *T_L is a sound and constructively complete tableau calculus for L and is compatible with* sub.

The theorem says that adding blocking as described above to any sound and complete tableau calculus for L turns it into a terminating calculus, when the properties stated in the theorem are true. This calculus then provides the basis for a tableau-based decision procedure for L.

The rest of this section is devoted to proving the theorem. In Sections 4 and 5 we show how the theorem can be applied to obtain tableau decision procedures for two description logics.

Proof. The blocking requirements (c1)–(c4) are sound in the sense that they cannot cause an open branch to become closed. The (ub) rule is sound in the usual sense. This implies that adding the blocking rule together with the requirements preserves soundness and (constructive) completeness. Hence $T_L +$ (ub) is sound and constructively complete.

Since $T_L +$ (ub) is compatible with sub the reason for non-termination is the possible infinite introduction of individuals in a branch. Because the ($\exists$) rule is the only rule that introduces new individuals (by assumption), limiting the application of this rule limits the length of derivations. This is what the next lemma says.

Lemma 1. *Let $\#^{\exists}(\mathcal{B})$ denote the number of applications of the ($\exists$) rule in a branch $\mathcal{B}$. If $\#^{\exists}(\mathcal{B})$ is finite then $\mathcal{B}$ is finite.*

Let C now be a fixed concept and suppose $\mathcal{B}$ is the *leftmost open branch with respect to* (ub) *branching* in the tableau $T_L(C)$. Let $\mathcal{I}(\mathcal{B})$ be an L-model of this branch; $\mathcal{I}(\mathcal{B})$ exists by constructive completeness of $T_L +$ (ub).

Lemma 2. *$\Delta^{\mathcal{I}(\mathcal{B})}$ is finite.*

Proof. Because L admits finite filtration there is an equivalence relation $\sim$ on $\Delta^{\mathcal{I}(\mathcal{B})}$ and a $\sim$-filtration $\overline{\mathcal{I}(\mathcal{B})}$ of $\mathcal{I}(\mathcal{B})$ which is a finite L-model. Let $\|\ell\|$ and $\|\ell'\|$ be any elements of $\mathcal{I}(\mathcal{B})$ such that $\|\ell\| \sim \|\ell'\|$. Let ℓ_0 and ℓ'_0 be the first individuals from the equivalence classes $\|\ell\|$ and $\|\ell'\|$, respectively, to which the (ub) rule has been applied. Let $\mathcal{S}$ be a segment of the branch $\mathcal{B}$ consisting of all concepts of the branch $\mathcal{B}$ strictly before the tableau node where the (ub) rule has been applied to these individuals. It can be shown that $\mathcal{S} \cup \{\ell_0 : \{\ell'_0\}\}$ is satisfiable in $\overline{\mathcal{I}(\mathcal{B})}$. Indeed, $\ell_0^{\overline{\mathcal{I}(\mathcal{B})}} = [\|\ell\|] = [\|\ell'\|] = \ell_0'^{\overline{\mathcal{I}(\mathcal{B})}}$. Because $T_L +$ (ub) is constructively complete, if $\ell : D \in \mathcal{S}$ then $\ell^{\mathcal{I}(\mathcal{B})} \in D^{\mathcal{I}(\mathcal{B})}$ and, hence, $\ell^{\overline{\mathcal{I}(\mathcal{B})}} \in D^{\overline{\mathcal{I}(\mathcal{B})}}$. We can see that compatibility of $T_L +$ (ub) with sub implies that $\overline{\mathcal{I}(\mathcal{B})}$ is a model of the signature $\mathsf{sub}(S)$. By soundness of $T_L +$ (ub) every satisfiable set has an open branch. Hence, there is an open left branch $\mathcal{B}'$ through the tableau node where the (ub) rule has been applied to ℓ_0 and ℓ'_0. Since $\mathcal{B}$ is already the leftmost open branch w.r.t. (ub), $\mathcal{B}'$ coincides with $\mathcal{B}$. That is, $\ell_0 : \{\ell'_0\} \in \mathcal{B}$ and $\|\ell\| = \|\ell'\|$. Therefore, $\Delta^{\mathcal{I}(\mathcal{B})} = \Delta^{\overline{\mathcal{I}(\mathcal{B})}}$ is finite.

Lemma 3. *Let $\|\ell\| \in \Delta^{\mathcal{I}(\mathcal{B})}$ be arbitrary, and let $\#^{\exists}(\|\ell\|)$ denote the number of applications of the ($\exists$) rule to concepts of the form $\ell' : \exists R.D$ with $\ell' \in \|\ell\|$. Then, $\#^{\exists}(\|\ell\|)$ is finite for every $\|\ell\| \in \Delta^{\mathcal{I}(\mathcal{B})}$.*

Proof. Suppose not, i.e. suppose $\#^{\exists}(\|\ell\|)$ is infinite. Since T_L+(ub) is compatible with sub the number of concepts of the form $\exists R.D$ such that $\ell : \exists R.D$ for some $\ell \in O$ is in the branch is finitely bounded by the size of $\mathsf{sub}(C)$. By requirements (c1) and (c2) there is a sequence of individuals $\ell_0, \ell_1, \ldots$ such that every $\ell_i \in \|\ell\|$ and the ($\exists$) rule has been applied to concepts $\ell_0 : \exists R.D, \ell_1 : \exists R.D, \ldots$ for some $\exists R.D \in \mathsf{sub}(C)$. However, such a situation is impossible because of requirements (c4) and (c3). For, without loss of generality we can assume that $\ell < \ell_0 < \ell_1 < \cdots$. Then, by requirement (c4), starting from some node of $\mathcal{B}$, as soon as ℓ_i appears in $\mathcal{B}$, it is detected that $\ell_i \in \|\ell\|$ before any next application of the ($\exists$) rule and, hence, ℓ_i is immediately blocked for any application of the ($\exists$) rule, by requirement (c3).

We have that $\#^{\exists}(\mathcal{B}) \leq \max\{\#^{\exists}(\|\ell\|) \mid \|\ell\| \in \Delta^{\mathcal{I}(\mathcal{B})}\} \times \mathsf{Card}(\Delta^{\mathcal{I}(\mathcal{B})})$. Hence, because $\Delta^{\mathcal{I}(\mathcal{B})}$ is finite and by Lemma 3 we obtain that $\#^{\exists}(\mathcal{B})$ is finite whenever $\mathcal{B}$ is the leftmost open branch with respect to the (ub) rule of a T_L+(ub)-tableau.

Termination is a consequence, because every closed branch of a $T_L +$ (ub)-tableau is trivially finite and the length of the leftmost open branch with respect to the (ub) rule is finite, too.

This completes the proof of Theorem 2.

3 Preliminaries for Applications

For the purpose of defining equivalence relations for filtrations in the next two sections we define the notions of unary and binary types. A *(unary) type* of an element x of an L-model $\mathcal{I}$ of a given signature of Σ, denoted by $\tau^{\Sigma}(x)$, is a set of all concept of the signature whose interpretations in $\mathcal{I}$ contain x, i.e., $\tau^{\Sigma}(x) \overset{\mathsf{def}}{=} \{C \in \Sigma \mid x \in C^{\mathcal{I}}\}$. A *(binary) type* of a pair (x, y) of elements of a L-model $\mathcal{I}$ of a signature of Σ, denoted by $\tau^{\Sigma}(x, y)$, is a set of all roles of the signature Σ which interpretations in $\mathcal{I}$ contain (x, y), i.e. $\tau^{\Sigma}(x, y) \overset{\mathsf{def}}{=} \{R \in \Sigma \mid (x, y) \in R^{\mathcal{I}}\}$.

4 A Mainstream Description Logic with Transitive Roles

In this section we show how the method can be applied to the description logic $\mathcal{SO}$, i.e. $\mathcal{ALCO}$ in which roles can be transitive.

The syntax and semantics of $\mathcal{SO}$ is a suitable extension of the syntax and semantics of $\mathcal{ALCO}$ in which a subset Trans of the atomic roles are transitive. The semantics of transitive roles is defined as expected: For every $s \in \mathsf{Trans}$, in every $\mathcal{SO}$-model $\mathcal{I}$, $s^{\mathcal{I}}$ is a transitive relation, i.e. for every $x, y, z \in \Delta^{\mathcal{I}}$, $(x, y) \in s^{\mathcal{I}}$ and $(y, z) \in s^{\mathcal{I}}$ implies $(x, z) \in s^{\mathcal{I}}$.

Let, for any set of concepts S, $\mathsf{sub}(S)$ consists of all concepts D such that D is a subconcept of some concept in S. The operator sub is trivially finite.

It is not difficult to see (or one can refer to standard filtration arguments for the multi-modal logic $\mathsf{S4}_{(m)}$,[2] cf. e.g. [3]) that $\mathcal{SO}$ admits finite filtration with respect to the equivalence $\sim$ defined by $x \sim y \overset{\mathsf{def}}{\iff} \tau^{\Sigma}(x) = \tau^{\Sigma}(y)$ for every $x, y \in \Delta^{\mathcal{I}}$, for a fixed finite signature Σ and a $\mathcal{SO}$-model $\mathcal{I}$ of the signature. An interpretation of every role r in the required finite $\sim$-filtration $\overline{\mathcal{I}}$ of $\mathcal{I}$ is defined by $([x],[y]) \in r^{\overline{\mathcal{I}}}$ iff $y \in C^{\mathcal{I}}$ implies $x \in (\exists r.C)^{\mathcal{I}}$ for every $\exists r.C \in \Sigma$.

A tableau calculus which is sound and complete for $\mathcal{SO}$, is the calculus comprising the $\mathcal{ALCO}$ rules of Figure 1 and the following rule, for each transitive role $s \in \mathsf{Trans}$:

$$(\mathsf{Trans}_s)\colon\ \frac{\ell : \exists s.\{\ell'\},\quad \ell' : \exists s.\{\ell''\}}{\ell : \exists s.\{\ell''\}}.$$

We use the notation $T_{\mathcal{SO}}$ to refer to this calculus. $T_{\mathcal{SO}}$ is not yet terminating; the concept $\exists s.p \sqcap \neg\exists s.\neg\exists s.p$ with $s \in \mathsf{Trans}$ gives a counter-example.

Now, it is not difficult to prove soundness in the usual way. The standard way of proving completeness also establishes constructive completeness of $T_{\mathcal{SO}}$. It can be seen that $T_{\mathcal{SO}}$ is compatible with sub. Each of the conditions of Theorem 2 hold therefore. As a consequence we get:

Theorem 3. *The calculus $T_{\mathcal{SO}}$ + (ub) is sound, complete, and terminating.*

This states that $T_{\mathcal{SO}}$ plus the unrestricted blocking rule is a terminating tableau and provides a decision procedure for $\mathcal{SO}$.

In $T_{\mathcal{SO}}$ the (Trans_s) rules capture transitivity directly. Alternatively, propagation rules may be used, which are common in tableau approaches for description logics and modal logics [4,5]. All the necessary results for calculi with propagation rules can also be proved in the presented framework. (It should be noted however that propagation rules approach is in general not sufficient for obtaining complete tableau calculi for description logics with role negation.)

5 A Description Logic with Boolean Role Operators

In this section we consider the description logic $\mathcal{ALBO}$. Finding the necessary constructions and giving the necessary proofs in order to establish the conditions of Theorem 2 is more involved than for $\mathcal{SO}$ in the previous section.

Syntax and semantics. $\mathcal{ALBO}$ extends $\mathcal{ALCO}$ with role operators, namely the Boolean operators (union and negation) and also role inverse. That is, roles are no longer just atomic roles (as in $\mathcal{SO}$), but are defined by this production rule:

$$R, S \overset{\mathsf{def}}{=} r \mid R^{-1} \mid \neg R \mid R \sqcup S.$$

As before r ranges over the set of atomic roles R.

[2] Adding nominals does not pose a problem.

The semantics of $\mathcal{ALBO}$ is given by a *model* (*interpretation*), as before, which is a tuple $\mathcal{I} = (\Delta^\mathcal{I}, C_0^\mathcal{I}, \ldots, \ell_0^\mathcal{I}, \ldots, R_0^\mathcal{I}, \ldots)$, except that the following additional conditions must hold:

$$(R^{-1})^\mathcal{I} = (R^\mathcal{I})^{-1}, \qquad (\neg R)^\mathcal{I} = (\Delta^\mathcal{I} \times \Delta^\mathcal{I}) \setminus R^\mathcal{I}, \qquad (R \sqcup S)^\mathcal{I} = R^\mathcal{I} \cup S^\mathcal{I}.$$

If $\mathcal{I}$ satisfies all the these conditions but possibly not $(\neg R)^\mathcal{I} \subseteq (\Delta^\mathcal{I} \times \Delta^\mathcal{I}) \setminus R^\mathcal{I}$, then we say $\mathcal{I}$ is a *quasi-model*. In other words, in any quasi-model, the inclusion $(\neg R)^\mathcal{I} \supseteq (\Delta^\mathcal{I} \times \Delta^\mathcal{I}) \setminus R^\mathcal{I}$ still holds but the reverse inclusion may fail.

In what follows, we prove that $\mathcal{ALBO}$ is complete with respect to the semantics of models, as well as the semantics of quasi-models, and define two sound, complete, and terminating tableau calculi for $\mathcal{ALBO}$. One calculus generates models, the other one generates quasi-models.

Closure operator. Let $\prec$ be the smallest transitive ordering on the set of all $\mathcal{ALBO}$ expressions (concepts and roles) satisfying:

(s1) $C \prec \neg C$,	(s6) $C \prec \exists R.C$,	(s11) $S \prec R \sqcup S$,
(s2) $C \prec C \sqcup D$,	(s7) $R \prec \exists R.C$,	(s12) $R \prec R^{-1}$,
(s3) $D \prec C \sqcup D$,	(s8) $\neg C \prec \neg\exists R.C$,	(s13) $R \prec \neg R$.
(s4) $\neg C \prec \neg(C \sqcup D)$,	(s9) $R \prec \neg\exists R.C$,	
(s5) $\neg D \prec \neg(C \sqcup D)$,	(s10) $R \prec R \sqcup S$,	

One can prove that $\prec$ is a well-founded ordering. Let $\preceq$ be the reflexive closure of $\prec$. For every set of concepts S we define $\mathsf{sub}(S) \overset{\text{def}}{=} \{E \mid E \preceq C \text{ for some } C \in S\}$. Because $\prec$ is well-founded, sub is a finite closure operator.

Nice filtration. Let Σ be a finite signature with respect to sub. Let $\sim$ be an equivalence relation on $\Delta^\mathcal{I}$ satisfying the following: for any $x, y, x', y' \in \Delta^\mathcal{I}$

(f1) if $x \sim y$ then $\tau^\Sigma(x) = \tau^\Sigma(y)$, and
(f2) if $x \sim y$ and $x' \sim y'$ then $\tau^\Sigma(x, x') = \tau^\Sigma(y, y')$.

Let $\overline{\mathcal{I}}$ be the $\sim$-filtration of $\mathcal{I}$ satisfying $R^{\overline{\mathcal{I}}} = \{([x], [y]) \mid \exists x' \sim x \exists y' \sim y\, (x', y') \in R^\mathcal{I}\}$ for every role R. It follows from (f2), that $R^{\overline{\mathcal{I}}} = \{([x], [y]) \mid (x, y) \in R^\mathcal{I}\}$ for every $R \in \Sigma$. By induction on the ordering $\prec$ it can be proved that all the conditions to be an $\mathcal{ALBO}$-model of the signature Σ are satisfied for $\overline{\mathcal{I}}$.

Thus, in order to prove that $\mathcal{ALBO}$ admits finite filtration we only need to give a definition of an equivalence which satisfies conditions (f1) and (f2) and results in a finite filtration $\overline{\mathcal{I}}$. However, because of condition (f2), it is not immediate how one could define the equivalence $\sim$ with a finite number of equivalence classes in the original model $\mathcal{I}$.

Standard filtration. Suppose now that the equivalence $\simeq$ is an equivalence relation which satisfies only the condition (f1), i.e. $x \simeq y$ implies $\tau^\Sigma(x) = \tau^\Sigma(y)$. The equivalence class $\lfloor x \rfloor$ for the element x from $\Delta^\mathcal{I}$ is defined as usual: $\lfloor x \rfloor \overset{\text{def}}{=} \{y \in \Delta^\mathcal{I} \mid x \simeq y\}$. We define a $\simeq$-filtration $\underline{\mathcal{I}}$ of $\mathcal{I}$ which is a quasi-model by

$$R^{\overline{\mathcal{I}}} \overset{\text{def}}{=} \{(\lfloor x \rfloor, \lfloor y \rfloor) \mid \exists x' \simeq x \exists y' \simeq y\, (x', y') \in R^\mathcal{I}\}.$$

It can be checked that $\underline{\mathcal{I}}$ is a quasi-model. In general, it is not a standard model however.

Conflict elimination process. Assume now that Σ is finite and *does not contain expressions with individual symbols*. That is, we focus first on $\mathcal{ALB}$. By a modification of the conflict elimination method for Boolean Modal Logic introduced by Gargov, Passy, and Tinchev [8] it is possible to transform a finite $\mathcal{ALB}$-quasi-model $\mathcal{I}$ into a finite $\mathcal{ALB}$-model $\mathcal{I}'$ and construct a p-morphism [3] f of the signature Σ from $\mathcal{I}'$ onto $\mathcal{I}$. In particular, we obtain that any concept D from Σ is satisfiable in $\mathcal{I}$ iff it is satisfiable in $\mathcal{I}'$.Summing up, we obtain the following theorem.

Theorem 4. *If C is a concept in the language of $\mathcal{ALB}$ and C is satisfiable in a quasi-model then it is satisfiable in a model. Furthermore, if C is satisfiable in a finite quasi-model then C is satisfiable in a finite model.*

Corollary 1.

1. *$\mathcal{ALB}$ is complete with respect to the class of all $\mathcal{ALB}$-quasi-models.*
2. *$\mathcal{ALB}$ admits finite filtration over the class of all $\mathcal{ALB}$-quasi-models.*

Finite nice filtration. Again, let Σ be a finite signature which does not contain any individuals. Having a finite model $\underline{\mathcal{I}}'$ for concept C and a p-morphism f, we can define an equivalence $\sim$ on $\mathcal{I}$ which satisfies conditions (f1) and (f2) so that the $\sim$-filtrated model $\overline{\mathcal{I}}$ is finite. Indeed, for all $x, y \in \Delta^{\mathcal{I}}$ we define

$$x \sim y \overset{\text{def}}{\iff} x \simeq y \text{ and for all } u, z \in \Delta^{\underline{\mathcal{I}}'} \text{ such that } f(u) = \lfloor x \rfloor = \lfloor y \rfloor,$$
$$\tau^{\Sigma}(u, z) = \tau^{\Sigma}(u, z) \text{ and } \tau^{\Sigma}(z, u) = \tau^{\Sigma}(z, u).$$

Because $\Delta^{\underline{\mathcal{I}}'}$ is finite and the number of unary types of a finite signature is finite, the number of equivalence classes with respect to $\sim$ in $\Delta^{\mathcal{I}}$ is finite. It can be straightforwardly checked that $\sim$ satisfies conditions (f1) and (f2).

Adding individuals. Because singleton interpretations of the concepts are preserved under filtration, we can replace, in the concept C, all singleton concepts with atomic concepts and, after construction of the required filtration, restore them. This extends the results above to $\mathcal{ALBO}$. Thus, the following theorem holds.

Theorem 5.

1. *$\mathcal{ALBO}$ is complete with respect to the class of all $\mathcal{ALBO}$-quasi-models.*
2. *$\mathcal{ALBO}$ admits finite (standard) filtration over the class of all $\mathcal{ALBO}$-quasi-models.*
3. *$\mathcal{ALBO}$ admits finite (nice) filtration (over the class of all $\mathcal{ALBO}$-models).*

Finally, from Lemma 1 we obtain the finite model property for $\mathcal{ALBO}$.

$$(\exists\sqcup):\ \frac{\ell : \exists(R \sqcup S).\{\ell'\}}{\ell : \exists R.\{\ell'\} \mid \ell : \exists S.\{\ell'\}} \qquad (\neg\exists\sqcup):\ \frac{\ell : \neg\exists(R \sqcup S).C}{\ell : \neg\exists R.C,\quad \ell : \neg\exists S.C}$$

$$(\exists^{-1}):\ \frac{\ell : \exists R^{-1}.\{\ell'\}}{\ell' : \exists R.\{\ell\}} \qquad (\neg\exists^{-1}):\ \frac{\ell : \neg\exists R^{-1}.C,\quad \ell' : \exists R.\{\ell\}}{\ell' : \neg C}$$

$$(\exists\neg):\ \frac{\ell : \exists\neg R.\{\ell'\}}{\ell : \neg\exists R.\{\ell'\}} \qquad (\neg\exists\neg):\ \frac{\ell : \neg\exists\neg R.C,\quad \ell' : \{\ell'\}}{\ell : \exists R.\{\ell'\} \mid \ell' : \neg C}$$

Fig. 2. Role decomposition rules in the tableau calculus $T_{\mathcal{ALBO}}$ for $\mathcal{ALBO}$

Theorem 6 (Finite Model Property of $\mathcal{ALBO}$). *$\mathcal{ALBO}$ has the finite model property, i.e., if a concept C is satisfiable, then it has a finite model.*

Tableau calculi. Because $\mathcal{ALBO}$ admits finite filtrations over both the class of $\mathcal{ALBO}$-models and the class of $\mathcal{ALBO}$-quasi-models, there are two ways of formulating a tableau calculus for $\mathcal{ALBO}$. Let $T_{\mathcal{ALBO}}$ be the calculus which extends the calculus $T_{\mathcal{ALCO}}$ (see Figure 1) with the rules listed in Figure 2. Let $T^q_{\mathcal{ALBO}}$ be the calculus consisting of all the rules of $T_{\mathcal{ALBO}}$ but without the ($\exists\neg$) rule. (The 'q' in the name of $T^q_{\mathcal{ALBO}}$ refers to quasi-models.)

The rules in Figure 2 are the rules for decomposing complex role expressions. They can be divided into two groups: rules for positive existential role occurrences (on the left) and rules for negated existential role occurrences (on the right). The rules can be seen to be reflections of the semantics of the main premise, which is the left premise in each case. Observe that the rules for positive existential role occurrences are limited to role assertions. This is justified because of the presence of the ($\exists$) rule.

Tableau derivations are constructed as usual. Suppose C is a given concept and we are interested in the satisfiability of C. First, preprocessing is performed on C in order to move the role inverse operator inwards so that they occur immediately in front of atomic roles, by rewriting based on these equivalences from left to right: $(\neg R)^{-1} = \neg(R^{-1})$, $(R \sqcup S)^{-1} = R^{-1} \sqcup S^{-1}$, $(R^{-1})^{-1} = R$. The tableau rules are then applied to $\ell : C'$, where ℓ is a new individual and C' is the transformed concept.

Soundness and completeness. We turn to proving soundness and completeness of the calculi. Because every rule preserves the satisfiability of concept assertions, it is easy to see that $T_{\mathcal{ALBO}}$ and $T^q_{\mathcal{ALBO}}$ are both sound for $\mathcal{ALBO}$.

For proving completeness of the calculi, suppose that a tableau $T_{\mathcal{ALBO}}(C)$, resp. $T^q_{\mathcal{ALBO}}(C)$, has been constructed for the given concept C. Suppose it is an open tableau, and $\mathcal{B}$ is an open branch in it. A quasi-model $\mathcal{I}(\mathcal{B})$ for the satisfiability of C can be constructed as follows. Recall that $\ell \sim_{\mathcal{B}} \ell'$ iff $\ell : \{\ell'\} \in \mathcal{B}$ and $\|\ell\| = \{\ell' \mid \ell \sim_{\mathcal{B}} \ell'\}$. For every concept D, define $D^{\mathcal{B}} \overset{\text{def}}{=} \{\|\ell\| \mid \ell : D \in \mathcal{B}\}$. For every role R, define $R^{\mathcal{B}} \overset{\text{def}}{=} \{(\|\ell\|, \|\ell'\|) \mid \exists \ell_0 \sim_{\mathcal{B}} \ell\ \exists \ell'_0 \sim_{\mathcal{B}} \ell'\ \ell_0 : \exists R.\{\ell'_0\} \in \mathcal{B}\}$. The rules (sym), (mon), and (id) ensure that the definition of $\mathcal{I}(\mathcal{B})$ does not depend on representatives of the equivalence classes. For any $\ell \in \mathsf{O}$, $p \in \mathsf{C}$, and

$r \in \mathsf{R}$ we set

$$\begin{aligned} \Delta^{\mathcal{I}(\mathcal{B})} &\overset{\text{def}}{=} \{\|\ell\| \mid \ell : \{\ell\} \in \mathcal{B}\}, \\ p^{\mathcal{I}(\mathcal{B})} &\overset{\text{def}}{=} p^{\mathcal{B}}, \\ r^{\mathcal{I}(\mathcal{B})} &\overset{\text{def}}{=} r^{\mathcal{B}}, \end{aligned} \qquad \ell^{\mathcal{I}(\mathcal{B})} \overset{\text{def}}{=} \begin{cases} \|\ell\|, & \text{if } \ell : \{\ell\} \in \mathcal{B}, \\ \|\ell'\|, & \text{for some } \|\ell'\| \in \Delta^{\mathcal{I}(\mathcal{B})}, \\ & \qquad\qquad \text{otherwise.} \end{cases}$$

The interpretations of any concept D and role R are defined inductively on the ordering $\prec$ by:

$$\begin{aligned} (D_0 \sqcup D_1)^{\mathcal{I}(\mathcal{B})} &\overset{\text{def}}{=} (D_0 \sqcup D_1)^{\mathcal{B}} \cup D_0^{\mathcal{I}(\mathcal{B})} \cup D_1^{\mathcal{I}(\mathcal{B})}, \\ (\neg D)^{\mathcal{I}(\mathcal{B})} &\overset{\text{def}}{=} (\neg D)^{\mathcal{B}} \cup (\Delta^{\mathcal{I}(\mathcal{B})} \setminus D^{\mathcal{I}(\mathcal{B})}), \\ (R_0 \sqcup R_1)^{\mathcal{I}(\mathcal{B})} &\overset{\text{def}}{=} (R_0 \sqcup R_1)^{\mathcal{B}} \cup R_0^{\mathcal{I}(\mathcal{B})} \cup R_1^{\mathcal{I}(\mathcal{B})}, \\ (\neg R)^{\mathcal{I}(\mathcal{B})} &\overset{\text{def}}{=} (\neg R)^{\mathcal{B}} \cup (\Delta^{\mathcal{I}(\mathcal{B})} \setminus R^{\mathcal{I}(\mathcal{B})}), \\ (R^{-1})^{\mathcal{I}(\mathcal{B})} &\overset{\text{def}}{=} (R^{-1})^{\mathcal{B}} \cup (R^{\mathcal{I}(\mathcal{B})})^{-1}. \end{aligned}$$

The following lemma can be proved by induction on the ordering $\prec$.

Lemma 4. *$\mathcal{I}(\mathcal{B})$ is an $\mathcal{ALBO}$-quasi-model. Moreover, if $\mathcal{B}$ is an open branch in $T_{\mathcal{ALBO}}$ then $\mathcal{I}(\mathcal{B})$ is an $\mathcal{ALBO}$-model.*

The constructive completeness of the calculi is an immediate consequence of this lemma.

Theorem 7.

1. *$T_{\mathcal{ALBO}}$ is sound and constructively complete with respect to $\mathcal{ALBO}$-models.*
2. *$T^q_{\mathcal{ALBO}}$ is sound and constructively complete with respect to $\mathcal{ALBO}$-quasi-models.*

Applying Theorem 2 together with Theorems 5 and 7 we obtain two sound, complete, and terminating calculi for $\mathcal{ALBO}$.

Theorem 8. *Both $T_{\mathcal{ALBO}}$ + (ub) and $T^q_{\mathcal{ALBO}}$ + (ub) are sound, complete, and terminating tableau calculi for $\mathcal{ALBO}$.*

The difference between the two calculi is that the calculus with the ($\exists\neg$) rule returns standard models while the other calculus returns only quasi-models.

6 Discussion

In this paper we have introduced a general method for turning ground semantic tableau calculi into decision procedures, and illustrated it with two examples.

The case of $\mathcal{SO}$ (i.e. $\mathcal{ALCO}$ with transitive roles) required a standard filtration argument which can be found in many modal logic textbooks, and the formulation of a sound and constructively complete tableau calculus which also does not pose any problems. The case of $\mathcal{ALBO}$ was more complicated. As should be expected, the most problematic part was to construct a finite filtration. In order to do this, we applied a 'simple' filtration argument to an $\mathcal{ALBO}$-model $\mathcal{I}$ which gave us a quasi-model which we then repaired using a modification of the method of Gargov, Passy, and Tinchev [8]. This produces a finite $\mathcal{ALB}$-model $\mathcal{I}'$. Using the obtained $\mathcal{ALB}$-model $\mathcal{I}'$, we defined a filtration of the original $\mathcal{ALBO}$-model which is finite because $\mathcal{I}'$ is finite. As a side effect we obtained completeness of $\mathcal{ALBO}$ with respect to quasi-models. The formulation of tableau rules which simply follow the semantics of the operators in $\mathcal{ALBO}$ was not difficult. Neither was obtaining the soundness and completeness results of the two tableau calculi, $T_{\mathcal{ALBO}}$ and $T^q_{\mathcal{ALBO}}$. The hard part would have been to turn the calculi into decision procedures and prove that the procedures are indeed terminating. Fortunately, Theorem 2 covers both cases, thus limiting the difficult part of the proof to the filtration argument.

The work builds on our previous work in [14], where we described a tableau-based decision procedure for $\mathcal{ALBO}$. In the current paper we have refined the techniques and presented a general method for proving the finite model property and developing tableau decision procedures. At the same time we have improved the results in a number of ways. The tableau calculi for $\mathcal{ALCO}$ and $\mathcal{ALBO}$ relative to standard models are almost identical to the tableau calculi of [14]; the only essential[3] difference is that the calculi in [14] include this additional congruence rule:

$$\textsf{(bridge)}: \frac{\ell : \exists R.\{\ell'\}, \quad \ell' : \{\ell''\}}{\ell : \exists R.\{\ell''\}}.$$

The results is this paper show that this rule is superfluous for completeness of $T_{\mathcal{ALCO}}$ and $T^q_{\mathcal{ALBO}}$. By the results in [14] the (bridge) rule is admissible in both and decidability is not harmed by the inclusion of it. New in the current paper is the tableau calculus $T^q_{\mathcal{ALBO}}$. It shows that the ($\exists\neg$) rule is not needed for completeness or decidability. The $T^q_{\mathcal{ALBO}}$ calculus constructs quasi-models. Sometimes it is not possible to find filtration arguments that yield standard models but filtration arguments can nevertheless be found for quasi-models [6]. Thus, being able to follow the approach using quasi-models means that it may still possible to devise decision procedures, even when the approach fails, or is difficult, for standard models.

The main result (Theorem 2) of the paper exploits similarities between filtration proofs and proofs of termination of tableau procedures. As said, it shifts the most difficult part of proving termination to finding a finite filtration. Filtration is a standard technique which is very powerful and well studied (cf. e.g. filtration proofs in [3,6,7,10]). Furthermore, every filtration depends only on the logic under consideration and its model theory, but does not depend on any particular tableau calculus. Hence, there is a separation between the computational machinery of the given logic and its logical properties, resulting in enhanced generality and flexibility of the framework. In particular, this makes it easier to develop refinements of tableau decision procedures devised within the framework

[3] In [14] the definition of $\mathcal{ALBO}$ is slightly different but definitionally equivalent and accordingly there is a small variation in the calculus.

through, for example, the incorporation of refinements of the rules, strategies, intelligent backtracking techniques and other optimisations.

The unrestricted blocking rule provides a simple way of doing blocking with backtracking. Backtracking is generally expensive but the rule can be flexibly refined so that backtracking is reduced or completely avoided. Imposing appropriate constraints on the unrestricted blocking rule, and using suitable rule application strategies, other blocking techniques [2,12] such as subset blocking, equality blocking, dynamic blocking (including pairwise blocking), static blocking, successor blocking, anywhere blocking and their combinations can be simulated. For instance, subset blocking can be implemented by allowing the application of the (ub) rule only when a subset check of the sets of concepts associated with the two individuals ℓ and ℓ' succeeds. For a logic with the tree-model property, one can add the constraint that ℓ is a successor of ℓ'; this simulates successor blocking.

The framework therefore provides a basis for enhancing prover engineering platforms such as those of [11,9,1] with a flexible blocking mechanism with which more general tableau decision procedures can then be constructed. The approach also provides the theoretical background for the way blocking is implemented in the MetTeL system [15]. MetTeL is a flexible engineering solution implemented in Java. It is intended to give easy and quick implementations of tableau procedures for a large class of logics (originally for logics of metrics and topology). In MetTeL, blocking could already be done universally and automatically for many logics, but the unrestricted blocking mechanism extends it further to decide logics with Boolean role operators which are currently outside scope of other similar systems and tableau provers.

Our approach is not tied to a specific description logic language. It can handle most familiar role constructors including union, intersection, composition, inverse, etc. Since the method is formulated for logics with individuals it can be applied to description logics with ABoxes. The main result also covers logics with general TBoxes and RBoxes. Implicit in our definitions is the assumption that operators additional to those of $\mathcal{ALCO}$ must include only role constructors with role arguments, but this is not essential. The assumption was only made for reasons of simplicity. In particular, the method can be easily extended to logics with arbitrary additional role and concept constructors including operators in which concepts and roles interact. Examples that come to mind are: test operator, image operator, sufficiency operators, universal modality, cross product, domain and range restriction; in fact most of these are already expressible in $\mathcal{ALBO}$ [14].

The conditions of the main Theorem 2 cover almost all known decidable description logics, and also modal logics, hybrid logics and even certain first-order fragments. For instance, because $\mathcal{ALBO}$ subsumes the two-variable fragment of the first-order logic the tableau calculi for $\mathcal{ALBO}$ presented in this paper can be transformed into ground tableau calculi for this fragment. This solves a long-standing open problem because so far it has not been known how to use tableau methods to solve the two-variable fragment.

The tableau rules are formalised in a generic way to cover most of the labelled and hybrid tableau formalisms. The restriction that (∃) is the only individual generating rule can be omitted and would then require an appropriate reformulation of conditions (c2), (c3), and (c4) of the blocking mechanism. In fact, the tableau calculus for the given logic can contain arbitrary individual generating rules. The specific kind of tableau rules used in the paper is not essential. It is possible to use other kinds of rules and tableau approaches, provided these satisfy the conditions of Theorem 2. For instance, the results transfer also to labelled KE tableau, ground hyperresolution with splitting, ground hypertableaux, and other bottom-up model generation procedures. We expect the techniques to be applicable more widely, for developing decision procedures based on other kinds of deduction methods and for much larger classes of logics and first-order fragments.

References

1. Abate, P., Goré, R.: The Tableaux Work Bench. In: Cialdea Mayer, M., Pirri, F. (eds.) TABLEAUX 2003. LNCS, vol. 2796, pp. 230–236. Springer, Heidelberg (2003)
2. Baader, F., Sattler, U.: An overview of tableau algorithms for description logics. Stud. Log. 69, 5–40 (2001)
3. Blackburn, P., de Rijke, M., Venema, Y.: Modal Logic. Camb. Tracts Theor. Comput. Sci, vol. 53. Cambridge University Press, Cambridge (2001)
4. Castilho, M.A., Fariñas del Cerro, L., Gasquet, O., Herzig, A.: Modal tableaux with propagation rules and structural rules. Fundam. Inform. 3-4(32), 281–297 (1997)
5. Fariñas del Cerro, L., Gasquet, O., Herzig, A., Sahade, M.: Modal tableaux: Completeness vs. termination. In: We Will Show Them!, vol. 1, pp. 587–614. College Publ. (2005)
6. Gabbay, D.M., Kurucz, A., Wolter, F., Zakharyaschev, M.: Many-Dimensional Modal Logics: Theory and Applications. North-Holland, Amsterdam (2003)
7. Gabbay, D.M., Shehtman, V.: Products of modal logics, part 1. Log. J. IGPL 6(1), 73–146 (1998)
8. Gargov, G., Passy, S., Tinchev, T.: Modal environment for Boolean speculations. In: Proc. Gödel 1986, Plenum, pp. 253–263 (1987)
9. Gasquet, O., Herzig, A., Sahade, M.: Terminating modal tableaux with simple completeness proof. In: Proc. AiML 2006, pp. 167–186. College Publ. (2006)
10. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press, Cambridge (2000)
11. Heuerding, A., Jäger, G., Schwendimann, S., Seyfried, M.: The Logics Workbench LWB: A snapshot. Euromath. Bull. 2(1), 177–186 (1996)
12. Horrocks, I., Sattler, U.: A description logic with transitive and inverse roles and role hierarchies. J. Log. Comput. 9(3), 385–410 (1999)
13. Schmidt, R.A., Tishkovsky, D.: A general tableau method for deciding description logics, modal logics and related first-order fragments, http://www.cs.man.ac.uk/~dmitry/papers/gtm2008.pdf
14. Schmidt, R.A., Tishkovsky, D.: Using tableau to decide expressive description logics with role negation. In: Aberer, K., Choi, K.-S., Noy, N., Allemang, D., Lee, K.-I., Nixon, L., Golbeck, J., Mika, P., Maynard, D., Mizoguchi, R., Schreiber, G., Cudré-Mauroux, P. (eds.) ISWC 2007. LNCS, vol. 4825, pp. 438–451. Springer, Heidelberg (2007)
15. Tishkovsky, D.: MetTeL system, http://www.cs.man.ac.uk/dmitry/implementations/MetTeL/

Terminating Tableaux for Hybrid Logic with the Difference Modality and Converse

Mark Kaminski and Gert Smolka

Programming Systems Lab, Saarland University,
Campus E1 3, 66123 Saarbrücken, Germany
{kaminski,smolka}@ps.uni-sb.de

Abstract. We present the first terminating tableau calculus for basic hybrid logic with the difference modality and converse modalities. The language under consideration is basic multi-modal logic extended with nominals, the satisfaction operator, converse, global and difference modalities. All of the constructs are handled natively.

To obtain termination, we extend chain-based blocking for logics with converse by a complete treatment of difference.

Completeness of our calculus is shown via a model existence theorem that refines previous constructions by distinguishing between modal and equational state equivalence.

Keywords: modal and hybrid logics, difference modality, converse, tableau systems, decision procedures.

1 Introduction

Modal logic with the difference modality $Dp = \lambda x.\, \exists y.\, x{\neq}y \wedge py$ is an expressive language [1,2]. It can express the global modality $Ep = p \,\dot{\vee}\, Dp$ and nominals $!p = E(p \,\dot{\wedge}\, \dot{\neg}(Dp))$. Gargov and Goranko [3] show that basic modal logic with D is equivalent with respect to modal definability to basic hybrid logic [2,4] with E (see also [5,6,7,8]).

Tableaux for modal logic with D are not well-understood. In a recent handbook chapter on modal proof theory [9], an unsound tableau calculus for basic modal logic with D is given.[1] A sound and complete tableau calculus for basic modal logic with D is given by Balbiani and Demri [1]. Unfortunately, Balbiani and Demri's calculus does not yield a decision procedure as it does not terminate on all inputs.

This paper presents a terminating prefixed tableau calculus for basic hybrid logic with D and converse. While it is possible to express the satisfaction operator @ and E in terms of D, it is more efficient to let the decision procedure handle satisfaction and global modalities natively. Hence, we allow @ and E as additional constructs in our language and extend the calculus to deal with them directly. So, the input language for our calculus is precisely characterized as basic

[1] The formula $\Box P \vee D\neg P$ is invalid but provable by the rules in [9].

multi-modal logic extended with nominals, the satisfaction operator, converse, global and difference modalities.

The first tableau-based decision procedure for a modal language extended with D as an additional operator was given in [10]. The blocking technique used there to ensure termination, called pattern-based blocking, is different from the traditional chain-based techniques [11,12,13,14] in that it does not exploit any information about the order in which prefixes are introduced to a tableau branch. In the presence of converse, however, pattern-based blocking as proposed in [10] is inherently incomplete.

Termination of the present calculus is obtained by chain-based blocking. Chain-based blocking was initially developed to deal with transitive modalities [11,12,13] and subsequently extended to logics with converse [15,16] and nominals [14,17]. As we show, the interaction between converse and D is similar to the interaction between converse and transitive modalities, and can be handled by adapting the techniques in [15,16] to account for the additional generative power of D.

Bolander and Blackburn [14] propose a different extension of chain-based blocking to global modalities and converse, blocking E by the same mechanism as diamond modalities. We propose an alternative treatment of global modalities. Besides, our approach differs from that of [14] in the model construction techniques employed to prove completeness of our calculus. As Bolander and Blackburn's approach does not cover D, they employ traditional filtration arguments, constructing a model that identifies prefixes modulo modal equivalence. To capture the semantics of D, we construct a model that does not necessarily identify modally equivalent prefixes, while still respecting the stronger equational equivalence.

Unlike our approach, which is cumulative and relies solely on tree-like structures, Horrocks and Sattler [17] propose a tableau calculus for a nominal logic with global modalities and converse based on possibly cyclic graph structures and treat equational equivalence by destructive graph transformation during tableau construction. Their calculus does not cover D but handles qualified number restrictions [18].

To treat D in a sound and complete way, the calculus by Balbiani and Demri [1] employs a computationally expensive cut rule. To avoid the general inefficiency coming with this rule, we follow [10] and integrate it into the rule for the dual of D. Thus the costs of the cut rule need only be paid if the dual of D is used.

It is possible to obtain decision procedures for the language under consideration by means of satisfiability-preserving translations into simpler languages [3,19,20,8] for which effective decision procedures are already available [21,14,10]. Our calculus yields the first effective decision procedure for modal logic with both D and converse modalities that does not rely on transformations of the input into other languages.

The paper is organized as follows. We start by formulating hybrid logic in simple type theory. Next, we present the rules of our calculus. Then, we impose control on the rules and show that the restricted calculus is terminating. The terminating calculus is then shown complete by means of a model existence theorem.

2 Hybrid Logic with D and Converse

We represent modal logic in simple type theory, which gives us an expressive syntax and a solid foundation. The basic idea of the representation goes back to Gallin [22] and can also be found in Gamut [23] (Sect. 5.8, two-sorted type theory). The representation of boxes and diamonds as higher-order constants appears in [24,25]. Since the type-theoretic representation formalizes the semantics of modal logic at the object level, one can prove meta- and object-level theorems of modal logic with a higher-order theorem prover [26].

We start with two base types B and S. The interpretation of B is fixed and consists of two truth values. The interpretation of S is a nonempty set whose elements are called worlds or states. Given two types σ and τ, the *functional type* $\sigma\tau$ is interpreted as the set of all total functions from the interpretation of σ to the interpretation of τ. We write $\sigma_1\sigma_2\sigma_3$ for $\sigma_1(\sigma_2\sigma_3)$.

We employ three kinds of variables: *Nominal variables* x, y, z of type S, *propositional variables* p, q of type SB, and *relational variables* r of type SSB. Nominal variables are called *nominals* for short. We use the logical constants

$$\begin{aligned} \bot, \top &: \mathrm{B} & \qquad \doteq &: \mathrm{SSB} \\ \neg &: \mathrm{BB} & \exists, \forall &: (\mathrm{SB})\mathrm{B} \\ \vee, \wedge, \rightarrow &: \mathrm{BBB} \end{aligned}$$

Terms are defined as usual. We write st for applications, $\lambda x.s$ for abstractions, and $s_1s_2s_3$ for $(s_1s_2)s_3$. We also use infix notation, e.g., $s \wedge t$ for $(\wedge)st$.

Terms of type B are called *formulas*. We employ some common notational conventions: $\exists x.s$ for $\exists(\lambda x.s)$, $\forall x.s$ for $\forall(\lambda x.s)$, and $x \dot{\neq} y$ for $\neg(x \doteq y)$.

The formulas of modal logic can be either translated to type-theoretic formulas (as in [22,23,25,27]) or directly represented as terms of type SB (as in [24,26,10]). Here we use the latter approach, which is more elegant since it models modal syntax directly as higher-order syntax. To do so, we need lifted versions of the Boolean connectives, which are defined as follows:

$$\begin{aligned} \dot{\neg} px &= \neg(px) & \qquad \dot{\neg} &: (\mathrm{SB})\mathrm{SB} \\ (p \,\dot{\wedge}\, q)x &= px \wedge qx & \dot{\wedge} &: (\mathrm{SB})(\mathrm{SB})\mathrm{SB} \\ (p \,\dot{\vee}\, q)x &= px \vee qx & \dot{\vee} &: (\mathrm{SB})(\mathrm{SB})\mathrm{SB} \end{aligned}$$

We can now write terms like $p \,\dot{\wedge}\, \dot{\neg} q$, which represent modal formulas. Here are the definitions of the remaining *modal constants* we will use:

$$\begin{aligned} r^{-}xy &= ryx & \qquad \_^{-} &: (\mathrm{SSB})\mathrm{SSB} \\ \langle r\rangle px &= \exists y.\, rxy \wedge py & \langle\_\rangle &: (\mathrm{SSB})(\mathrm{SB})\mathrm{SB} \\ [r]px &= \forall y.\, rxy \rightarrow py & [\_] &: (\mathrm{SSB})(\mathrm{SB})\mathrm{SB} \\ Epx &= \exists p & E &: (\mathrm{SB})\mathrm{SB} \\ Apx &= \forall p & A &: (\mathrm{SB})\mathrm{SB} \end{aligned}$$

$$\begin{aligned} Dpx &= \exists y.\, x \dot{\neq} y \wedge py & \qquad D &: (\mathrm{SB})\mathrm{SB} \\ \bar{D}px &= \forall y.\, x \dot{=} y \vee py & \bar{D} &: (\mathrm{SB})\mathrm{SB} \\ \dot{x}y &= x \dot{=} y & \dot{\_} &: \mathrm{SSB} \\ @xpy &= px & @ &: \mathrm{S}(\mathrm{SB})\mathrm{SB} \end{aligned}$$

Applied to a relation r, the operator $\_^-$ yields the converse of r. This allows us to add converse to our language without introducing converse versions of the operators $\langle \_ \rangle$ and $[\_]$. We call a term $t : \mathrm{SB}$ *modal* if it has the form

$$\begin{aligned} \rho &::= r \mid r^- \\ t &::= p \mid \dot{\neg} t \mid t \circ t \mid \mu\rho t \mid \nu t \mid \dot{x} \mid @xt \end{aligned}$$

where $\circ \in \{\dot{\wedge}, \dot{\vee}\}$, $\mu \in \{\langle \_ \rangle, [\_]\}$ and $\nu \in \{E, A, D, \bar{D}\}$.

A *modal interpretation* $\mathfrak{M}$ is an interpretation of simple type theory that interprets B as the set $\{0, 1\}$, $\bot$ as 0 (i.e., false), $\top$ as 1 (i.e., true), maps S to a non-empty set, gives the logical constants $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\exists$, $\forall$, $\dot{=}$ their usual meaning, and satisfies the equations defining the modal constants $\dot{\neg}$, $\dot{\wedge}$, $\dot{\vee}$, $\_^-$, $\langle \_ \rangle$, $[\_]$, E, A, D, $\bar{D}$, $\dot{\_}$, and @. Whenever $\mathfrak{M}t = 1$, we say that $\mathfrak{M}$ *satisfies* t, or that $\mathfrak{M}$ is a *model* of t. A formula is called *satisfiable* if it has a satisfying modal interpretation.

We now give some additional syntactic definitions that are needed for the rest of the paper. A modal term $s : \mathrm{SB}$ is called *normal* if it is in negation-normal form, that is, has the form

$$s ::= p \mid \dot{\neg} p \mid s \circ s \mid \mu\rho s \mid \nu s \mid \dot{x} \mid \dot{\neg}\dot{x} \mid @xs$$

where $\circ \in \{\dot{\wedge}, \dot{\vee}\}$, $\mu \in \{\langle \_ \rangle, [\_]\}$ and $\nu \in \{E, A, D, \bar{D}\}$. A formula s is called *normal* if it has the form tx where t is a normal modal term. Formulas of the form rxy or r^-xy are called *accessibility formulas* or *edges*.

Given a term t, we write $\mathcal{N}t$ for the set of nominals that occur in t. The notation is extended to sets of terms in the natural way: $\mathcal{N}X := \bigcup\{\mathcal{N}t \mid t \in X\}$.

3 Tableau Rules

Our tableaux are constructed in the usual way from a finite non-empty set of *initial* normal formulas by the rules in Fig. 1. The rules may extend tableau branches by formulas s of the form

$$s ::= x \dot{=} y \mid x \dot{\neq} y \mid \rho xy \mid tx$$

where t is a normal modal term. Single tableau branches are referred to by the meta-variables Γ and Δ. We allow no multiple occurrences of identical formulas on a single branch. Nominals x occurring in normal formulas sx are used to reference individual states, analogously to prefixes as used by related calculi [28, 14] and, similarly, prefixed calculi for nominal-free logics [9], with the important difference that in our case prefixes are part of the object language. We use edges to represent assertions about accessibility relations and equations for state equality or inequality constraints.

Given a branch Γ, we use $\sim_\Gamma$ to denote the equivalence closure of the relation $\{(x,y) \mid x \doteq y \in \Gamma\}$. If $x \sim_\Gamma y$, we call x and y *equationally equivalent* on Γ.

It is easy to verify that the rules in Fig. 1 are sound in the following sense.

Proposition 1 (Soundness). *Let Γ be a tableau branch and $\Delta_1, \dots, \Delta_n$ be the extensions of Γ obtained by a rule $\mathcal{R}$ from Fig. 1 ($n \in \{1,2\}$). Then Γ is satisfiable if and only if there is some $i \in \{1,\dots,n\}$ such that Δ_i is satisfiable.*

Unlike [28, 14] but similarly to [15, 17], we use signed edges of the form rxy and r^-xy. We define $\tilde{r} := r^-$ and $\widetilde{r^-} := r$. Semantically, rxy is considered identical to r^-yx. But the former formula additionally records that y was added to the branch after x, while the latter formula implies the converse. This way, we have an explicit representation of all the chronological information that will be necessary in Sect. 4 to impose a terminating control on the rules.

As all the relevant chronological information is contained in the edges, we can ignore the vertical structure of tableau branches and see them as sets of formulas, which may be subject to the usual set predicates and operators. For instance, we may write $s \in \Gamma$ to denote that s occurs on Γ, and $\Gamma - \Delta$ for the set of formulas that occur on Γ but not on Δ. The notation $\mathcal{N}\Gamma$ is defined in the obvious way.

$$\mathcal{R}_{\dot\wedge}\ \frac{(s \mathbin{\dot\wedge} t)x}{sx,\ tx} \qquad \mathcal{R}_{\dot\vee}\ \frac{(s \mathbin{\dot\vee} t)x}{sx \mid tx} \qquad \mathcal{R}_{\Diamond}\ \frac{\langle\rho\rangle tx}{\rho xy,\ ty}\ y \notin \mathcal{N}\Gamma \qquad \mathcal{R}_{\Box}\ \frac{[\rho]tx \qquad \rho xy}{ty}$$

$$\mathcal{R}_{\tilde\Box}\ \frac{[\rho]tx \qquad \tilde\rho yx}{ty} \qquad \mathcal{R}_{E}\ \frac{Etx}{ty}\ y \notin \mathcal{N}\Gamma \qquad \mathcal{R}_{A}\ \frac{Atx}{ty}\ y \in \mathcal{N}\Gamma$$

$$\mathcal{R}_{\doteq}\ \frac{sx}{sy}\ x \sim_\Gamma y,\ s \text{ modal} \qquad \mathcal{R}_{N}\ \frac{\dot{x}y}{x \doteq y} \qquad \mathcal{R}_{\bar N}\ \frac{\dot\neg \dot{x}y}{x \dot{\neq} y} \qquad \mathcal{R}_{@}\ \frac{@ytx}{ty}$$

$$\mathcal{R}_{D}\ \frac{Dtx}{x \dot{\neq} y,\ ty}\ y \notin \mathcal{N}\Gamma \qquad \mathcal{R}_{\bar D}\ \frac{\bar{D}tx}{x \doteq y \mid ty}\ y \in \mathcal{N}\Gamma$$

Γ is the tableau branch to which a rule is applied.

Fig. 1. Tableau Rules

We call a branch Γ *closed* if there is some p, x and y such that Γ contains either both px and $\dot\neg px$ or a disequation $x \dot{\neq} y$ where $x \sim_\Gamma y$. Otherwise, Γ is called *open*. A tableau is called closed if all of its branches are closed, and open otherwise. To prove a modal term s valid, one computes the negation-normal form t of $\dot\neg s$, selects a nominal $x \notin \mathcal{N}t$, and constructs a closed tableau for tx.

4 Control

It is easy to see that our tableau rules do not terminate without additional restrictions on their applicability. Figure 2 shows a possible non-terminating derivation. So, to achieve termination, we need to impose on our rules a terminating control.

$$\begin{array}{ll}
A(\langle r\rangle p)x & \\
\langle r\rangle px & \mathcal{R}_A \\
rxy,\ py & \mathcal{R}_\Diamond \\
\langle r\rangle py & \mathcal{R}_A \\
\dots &
\end{array}$$

Fig. 2. A Non-terminating Tableau Derivation

Every tableau branch Γ can be seen as a graph with the vertices $\mathcal{N}\Gamma$ and the edges given by the relation $<_\Gamma := \{(x,y) \mid \exists\rho\colon\ \rho xy \in \Gamma\}$. The relations $<_\Gamma^+$ and $<_\Gamma^*$ are defined from $<_\Gamma$ as usual (transitive and reflexive transitive closure). We define $G_\Gamma := (\mathcal{N}\Gamma, <_\Gamma)$.

A modal term s is said to *occur at a nominal x on a tableau branch Γ* if sx occurs on Γ. We define the *labeling* $\mathcal{L}_\Gamma x$ of a nominal x on a branch Γ to be set of all modal terms that occur at x on Γ. Two nominals x, y are called *modally equivalent* on a branch Γ if $\mathcal{L}_\Gamma x = \mathcal{L}_\Gamma y$. The function $\mathcal{L}_\Gamma$ defines a vertex labeling of G_Γ with sets of modal terms. We say a nominal x is a *root* of G_Γ if x has no predecessor in $<_\Gamma$, and write $\mathrm{Root}\,\Gamma$ for the set of all roots of G_Γ.

The graph G_Γ should *not* be understood as a partial model of Γ. So, the connection between $<_\Gamma$ and the transition relations in possible models of Γ is relatively loose. In particular, our tableau algorithm will always keep $<_\Gamma$ acyclic while actual models of Γ may contain cycles.

Achieving termination is easy once we can give an upper bound on the number of vertices in G_Γ. In particular, we would like to be able to bound the maximal length of chains $x_1 <_\Gamma \dots <_\Gamma x_n$. To do so, we want to avoid extending such chains if they are repeating, i.e., contain two distinct nominals with the same labeling. This motivates the following definition: A nominal x is called *active* on a branch Γ if there are no two distinct nominals $y, z <_\Gamma^* x$ such that $\mathcal{L}_\Gamma y = \mathcal{L}_\Gamma z$. Otherwise, x is called *inactive*.

We say a formula s is *expanded* on a branch Γ if one of the following *expandedness conditions* holds:

($\mathcal{E}_\wedge$) $s = (t_1 \dot\wedge t_2)x$ and $t_1x, t_2x \in \Gamma$
($\mathcal{E}_\vee$) $s = (t_1 \dot\vee t_2)x$ and $t_1x \in \Gamma$ or $t_2x \in \Gamma$
($\mathcal{E}_\Diamond$) $s = \langle\rho\rangle tx$ and there is some y such that $ty \in \Gamma$ and either $\rho xy \in \Gamma$ or $\tilde\rho yx \in \Gamma$
($\mathcal{E}_\Box$) $s = [\rho]tx$ and, for every y such that $\rho xy \in \Gamma$ or $\tilde\rho yx \in \Gamma$, it holds $ty \in \Gamma$

($\mathcal{E}_E$) $s = Etx$ and there is some $y \in \text{Root}\,\Gamma$ such that $ty \in \Gamma$
($\mathcal{E}_A$) $s = Atx$ and, for every $y \in \mathcal{N}\Gamma$, it holds $ty \in \Gamma$
($\mathcal{E}_{\dot=}$) $s = x\dot{=}y$ and $\mathcal{L}_\Gamma x = \mathcal{L}_\Gamma y$
($\mathcal{E}_N$) $s = \dot{y}x$ and $y\dot{=}x \in \Gamma$
($\mathcal{E}_{\bar{N}}$) $s = \dot\neg\dot{y}x$ and $y\dot{\neq}x \in \Gamma$
($\mathcal{E}_@$) $s = @ytx$ and $ty \in \Gamma$
($\mathcal{E}_D$) $s = Dtx$ and there is some $y \in \text{Root}\,\Gamma$ such that $y \not\sim_\Gamma x$ and $ty \in \Gamma$
($\mathcal{E}_{\bar{D}}$) $s = \bar{D}tx$ and, for every $y \in \mathcal{N}\Gamma$, either $x \sim_\Gamma y$ or $ty \in \Gamma$

Note that there are no expandedness conditions for formulas of the form px, $\dot\neg px$ and $x\dot{\neq}y$.

We restrict the applicability of our tableau rules by two conditions.

($\mathcal{C}_1$) A rule is applicable to a formula $s \in \Gamma$ only if Γ is open, s is not expanded on Γ, and if the rule application results in a *proper extension* of Γ, i.e., extends Γ by at least one formula that does not already occur on Γ.
($\mathcal{C}_2$) A rule is applicable to a formula of the form $\langle\rho\rangle tx$ on Γ only if x is active on Γ.

Note that $\mathcal{C}_1$ applies to all formulas, including diamonds, while $\mathcal{C}_2$ applies to diamond formulas only.

Except possibly for the cases $\mathcal{E}_E$ and $\mathcal{E}_D$, the condition $\mathcal{C}_1$ is intuitive. Indeed, similar conditions are often assumed implicitly when formulating tableau systems. The restriction $\mathcal{C}_2$ is a chain-based blocking condition as in [15,16].

Incidentally, $\mathcal{E}_\Diamond$ has a well-known analog in tableaux for classical first-order logic. There, the applicability of the existential rule δ can be restricted to once per formula. In a somewhat less obvious way, $\mathcal{E}_E$ and $\mathcal{E}_D$ also relate to this restriction. More details are provided later.

We are going to show that our calculus with the two applicability restrictions is complete and terminating, thus yielding a decision procedure for hybrid logic with D and converse. If a branch cannot be extended by any tableau rules, we call it *maximal*. Assuming that our calculus terminates, its completeness is proven by showing that an open and maximal tableau branch always exhibits a model of its initial formulas.

In the cases $\mathcal{E}_E$ and $\mathcal{E}_D$, it may seem unclear why we want the witness of s (i.e., the nominal y such that $ty \in \Gamma$) to be a root of G_Γ. One may consider taking the following weaker versions of $\mathcal{E}_E$ and $\mathcal{E}_D$:

($\mathcal{E}'_E$) Etx is expanded if there is some y such that $ty \in \Gamma$.
($\mathcal{E}'_D$) Dtx is expanded if there is some y such that $x \not\sim_\Gamma y$ and $ty \in \Gamma$.

It turns out, however, that if we do so, the interaction of $\mathcal{C}_1$ with $\mathcal{C}_2$ will render our calculus incomplete. Figure 3 shows an open branch for the unsatisfiable set $\{A(\langle r\rangle p)x, A(\langle r\rangle\dot\bot \dot\vee E(\langle r\rangle\dot\bot))x\}$, where $\dot\bot := q \dot\wedge \dot\neg q$, which becomes maximal if we weaken $\mathcal{E}_E$ to $\mathcal{E}'_E$. An example for $\mathcal{E}'_D$ looks analogously.

Another variant of $\mathcal{E}_D$ that we might consider corresponds more closely to the tableau rule for D:

($\mathcal{E}''_D$) Dtx is expanded if there is some y such that $x\dot{\neq}y, ty \in \Gamma$.

Here, it is termination that is no longer guaranteed, as shown in Fig. 4.

$A(\langle r\rangle p)x,\ A(\langle r\rangle\dot{\bot}\,\dot{\vee}\,E(\langle r\rangle\dot{\bot}))x$	
$\langle r\rangle px$	$\mathcal{R}_A$
$(\langle r\rangle\dot{\bot}\,\dot{\vee}\,E(\langle r\rangle\dot{\bot}))x$	$\mathcal{R}_A$
$E(\langle r\rangle\dot{\bot})x$	$\mathcal{R}_{\dot{\vee}}$
$rxy,\ py$	$\mathcal{R}_{\Diamond}$
$\langle r\rangle py$	$\mathcal{R}_A$
$(\langle r\rangle\dot{\bot}\,\dot{\vee}\,E(\langle r\rangle\dot{\bot}))y$	$\mathcal{R}_A$
$E(\langle r\rangle\dot{\bot})y$	$\mathcal{R}_{\dot{\vee}}$
$ryz,\ pz$	$\mathcal{R}_{\Diamond}$
$\langle r\rangle pz$	$\mathcal{R}_A$
$rzu,\ pu$	$\mathcal{R}_{\Diamond}$
$\langle r\rangle pu$	$\mathcal{R}_A$
$(\langle r\rangle\dot{\bot}\,\dot{\vee}\,E(\langle r\rangle\dot{\bot}))u$	$\mathcal{R}_A$
$\langle r\rangle\dot{\bot}u$	$\mathcal{R}_{\dot{\vee}}$
$(\langle r\rangle\dot{\bot}\,\dot{\vee}\,E(\langle r\rangle\dot{\bot}))z$	$\mathcal{R}_A$
$E(\langle r\rangle\dot{\bot})z$	$\mathcal{R}_{\dot{\vee}}$

Fig. 3. A Maximal Tableau Branch with the Expandedness Condition $\mathcal{E}'_E$

$A(D(D(Dp)))x$	
$D(D(Dp))x$	$\mathcal{R}_A$
$x\dot{\neq}y,\ D(Dp)y$	$\mathcal{R}_D$
$y\dot{\neq}z,\ Dpz$	$\mathcal{R}_D$
$z\dot{\neq}u,\ pu$	$\mathcal{R}_D$
$D(D(Dp))u$	$\mathcal{R}_A$
$\ldots$	

Fig. 4. A Non-terminating Tableau Derivation with the Expandedness Condition $\mathcal{E}''_D$

5 Termination

We will now show that every tableau derivation is finite. Since the two branching rules $\mathcal{R}_{\dot{\vee}}$ and $\mathcal{R}_{\bar{D}}$ are at most binary, by König's lemma it suffices to show that the length of the individual branches is bounded.

Since every rule application extends a branch only by formulas that do not yet occur on the branch, the length of a branch Γ coincides with the number of formulas on Γ. First, let us show that this number is bounded by a function in the number of nominals on Γ. Then, we will show that this number is itself bounded from above, completing the termination proof.

We write $\Gamma \to \Delta$ to denote that the branch Δ is an extension of a branch Γ obtained by a single rule application. The notations $\Gamma \to^{+} \Delta$ and $\Gamma \to^{*} \Delta$

are then defined in the obvious way. We write $\mathrm{Mod}\,\Gamma$ for the set of all modal terms occurring on Γ, possibly as subterms, and $\mathrm{Rel}\,\Gamma$ for the set of all relational variables that occur on Γ.

Crucial for our termination argument is the fact that our rules cannot introduce to the tableau any modal terms that do not already occur as subterms of the initial formulas.

Proposition 2 (Subterm Property). *If* $\Gamma \to^* \Delta$, *then* $\mathrm{Mod}\,\Gamma = \mathrm{Mod}\,\Gamma$.

For every pair of nominals x, y and every relation r, a branch Γ may contain edges rxy and $r^{-}xy$, equations $x \dot{=} y$, disequations $x \dot{\neq} y$ and, for every term $s \in \mathrm{Mod}\,\Gamma$, a formula sx. Hence, the size of Γ is bounded by $2|\mathrm{Rel}\,\Gamma| \cdot |\mathcal{N}\Gamma|^2 + 2|\mathcal{N}\Gamma|^2 + |\mathrm{Mod}\,\Gamma| \cdot |\mathcal{N}\Gamma|$. By Proposition 2, we know that $|\mathrm{Mod}\,\Gamma|$ and $|\mathrm{Rel}\,\Gamma|$ depend only on the initial formulas of the tableau.

So, it suffices to show that $|\mathcal{N}\Gamma|$ is bounded. We do so by showing that G_Γ is a finite forest of a size bounded by some function in the initial branch Γ_0. Looking at how Γ is constructed, it is easy to see that G_Γ is a well-founded forest, so it remains to show that:

1. Every tree in G_Γ has bounded outdegree.
2. Every tree in G_Γ has bounded depth.
3. G_Γ has a bounded number of roots.

The first bound is obtained by observing that edges are only added by the rule $\mathcal{R}_\Diamond$. It is easy to see that once $\mathcal{R}_\Diamond$ is applied to some formula s, s will be expanded on all extensions of the resulting branch. Hence, the outdegree of a nominal x is bounded by the number of distinct terms $\langle\rho\rangle t$ that occur at x, which, in its turn, is bounded by $|\mathrm{Mod}\,\Gamma_0|$.

The bound on the depth of the trees in G_Γ is $2^{|\mathrm{Mod}\,\Gamma_0|} + 1$, which easily follows from the fact that, by the Subterm Property and the pigeonhole principle, every sequence $x_1 <_\Gamma \ldots <_\Gamma x_{2^{|\mathrm{Mod}\,\Gamma_0|}+1}$ contains at least two distinct but modally equivalent nominals.

Now to the the number of roots in G_Γ. The applicability condition $\mathcal{C}_1$ enforces that the number of distinct formulas on a branch is strictly increased by every rule application.

Proposition 3. *If* $\Gamma \to^+ \Delta$, *then* $\Gamma \subsetneq \Delta$.

Note that since our tableaux are constructed starting from normal formulas, $<_{\Gamma_0}$ is always empty. Hence, since Γ_0 is non-empty, $\mathrm{Root}\,\Gamma$ contains at least one nominal. Moreover, whenever a branch Γ is extended by a formula ρxy, we require that $y \notin \mathcal{N}\Gamma$. Therefore, once a nominal is a root of Γ, it will remain a root for every extension of Γ.

Proposition 4. *If* $\Gamma \to^* \Delta$, *then* $\mathrm{Root}\,\Gamma \subseteq \mathrm{Root}\,\Delta$.

Since there are only two rules that can introduce new roots to G_Γ, namely $\mathcal{R}_E$ and $\mathcal{R}_D$, it suffices to show that the number of their applications in any derivation is bounded from above by a function in the initial branch Γ_0. The bound for $\mathcal{R}_E$ is given by $\mathcal{B}_E\Gamma_0$, and the bound for $\mathcal{R}_D$ by $\mathcal{B}_D\Gamma_0$, where $\mathcal{B}_E$ and $\mathcal{B}_D$ are defined as follows.

$$\mathcal{B}_E\Gamma \ := \ |\mathrm{Mod}\,\Gamma - \{\, s \mid \exists\, x \in \mathrm{Root}\,\Gamma\colon sx \in \Gamma \,\}|$$

The intuition behind $\mathcal{B}_E\Gamma$ is that $\mathcal{R}_E$ can only be applied once per modal term, independently of the nominal at which the term occurs. By Propositions 2, 3 and 4, $\mathcal{B}_E\Gamma$ is decreased by every application of $\mathcal{R}_E$ and not increased by any of the other rules. The definition

$$\begin{aligned}\mathcal{B}_D\Gamma := {} & |\mathrm{Mod}\,\Gamma - \{s \,|\, \exists\, y \in \mathrm{Root}\,\Gamma\colon sy \in \Gamma\,\}| \\ & + |\mathrm{Mod}\,\Gamma - \{s \,|\, \exists\, x \in \mathcal{N}\Gamma\ \exists\, y,z \in \mathrm{Root}\,\Gamma\colon x \sim_\Gamma y \text{ and } \{sy, x \dot{\neq} z, sz\} \subseteq \Gamma\,\}|\end{aligned}$$

follows the same idea, with the intuition here being that $\mathcal{R}_D$ is applicable at most twice per modal term. One can verify that $\mathcal{B}_D\Gamma$ is decreased by $\mathcal{R}_D$ and not increased by any of the other rules. That the second argument of the sum is needed can be seen with the branch $\{Dsx,\ x \dot{\neq} y,\ sy, Dsy\}$, where y is a root and Dsy is not expanded. To see that $\mathcal{R}_D$ is not applicable to a formula $Dsu \in \Gamma$ once, for some $x \in \mathcal{N}\Gamma$ and $y,z \in \mathrm{Root}\,\Gamma$, it holds $x \sim_\Gamma y$ and $\{sy,\ x \dot{\neq} z,\ sz\} \subseteq \Gamma$, observe that Dsx is expanded unless $x \sim_\Gamma y \sim_\Gamma z \sim_\Gamma u$, in which case Γ is closed and hence maximal.

6 Model Existence

To prove our calculus complete, it remains to show that every open maximal extension Γ of an initial branch Γ_0 exhibits a model $\mathfrak{M}$ of Γ_0. Without converse modalities, we can construct $\mathfrak{M}$ such that it satisfies not only Γ_0 but all formulas on Γ [28,14,10]. With converse, however, it seems easier to construct a model only for a distinguished subset X of Γ that still contains Γ_0. It is known [15,14] that the set of formulas occurring at nominals active on Γ is a suitable candidate for X.

The model construction by Bolander and Blackburn [14] deals with equational equivalence of nominals by identifying nominals up to modal equivalence (this approach is commonly known as filtration). Two nominals x and y are mapped to the same state if $\mathcal{L}_\Gamma x = \mathcal{L}_\Gamma y$. This suffices because on saturated branches equational equivalence implies modal equivalence. However, the approach is no longer appropriate once we extend our language by D. Look at the branch $\Gamma := \{Dpx, px, Dpy, py\}$. A model of Γ needs at least two different states, both of which may satisfy the same set of formulas. To avoid this problem, we base our model construction not on modal equivalence but directly on equational equivalence as defined by the relation $\sim_\Gamma$.

We proceed in several steps. Starting with a branch Γ, we apply to it a substitution φ eliminating syntactically distinct nominals that are equivalent modulo $\sim_\Gamma$. Then, we construct a model $\mathfrak{M}$ of a distinguished subset φX of $\varphi\Gamma$ such that X contains Γ_0. Finally, we show how to extend $\mathfrak{M}$ to a model of X.

A *nominal substitution* φ is a function $Nom \to Nom$, where Nom is the set of all nominals. Since nominal substitutions are the only kind of substitutions we will look at, in the following we will refer to them simply as "substitutions". We write φs for the term obtained by replacing every nominal x in s by φx. So, for instance, $\varphi((@x\dot{y})z) = (@x'\dot{y}')z'$ if $\varphi x = x'$, $\varphi y = y'$ and $\varphi z = z'$. Substitutions are extended to sets of terms in the intuitive way. Given a branch Γ, we call a substitution φ a *normalizer* for Γ if $\varphi x \sim_\Gamma x$ for all $x \in \mathcal{N}\Gamma$ and $\forall x, y \in \mathcal{N}\Gamma\colon\ \varphi x = \varphi y \iff x \sim_\Gamma y$. Note that, given an at most countable branch Γ, a normalizer φ for Γ can always be constructed by taking an arbitrary well-ordering $\prec$ of Γ and setting $\varphi := \{\, (x, y) \in (\mathcal{N}\Gamma)^2 \mid y = \min_\prec \{\, z \in \mathcal{N}\Gamma \mid x \sim_\Gamma z \,\}\,\}$. Hence, normalizers exist for every branch Γ of our calculus. They are not unique since neither are well-orderings of Γ.

Lemma 1. *Let Γ be open and maximal. If $x \sim_\Gamma y$, then $\mathcal{L}_\Gamma x = \mathcal{L}_\Gamma y$.*

Lemma 2. *Let Γ be open and maximal and φ a normalizer for Γ. If $\mathcal{L}_\Gamma x = \mathcal{L}_\Gamma y$, then $\mathcal{L}_{\varphi\Gamma}(\varphi x) = \mathcal{L}_{\varphi\Gamma}(\varphi y)$.*

Proof. Clearly, $\mathcal{L}_{\varphi\Gamma}(\varphi z) = \bigcup_{u \sim_\Gamma z} \varphi(\mathcal{L}_\Gamma u)$. By Lemma 1, the latter is the same as $\varphi(\mathcal{L}_\Gamma z)$. So, $\mathcal{L}_{\varphi\Gamma}(\varphi x) = \varphi(\mathcal{L}_\Gamma x) = \varphi(\mathcal{L}_\Gamma y) = \mathcal{L}_{\varphi\Gamma}(\varphi y)$. □

A nominal x is called *relevant* on Γ if every y such that $y <^+_\Gamma x$ is active.

Proposition 5. *Every nominal that is active on a branch Γ is relevant on Γ.*

Proposition 6. *If x is active on Γ and either $\rho xy \in \Gamma$ or $\rho yx \in \Gamma$, then y is relevant on Γ.*

Proposition 7. *If x is relevant on Γ, then there is some $y <^*_\Gamma x$ such that y is active on Γ and $\mathcal{L}_\Gamma y = \mathcal{L}_\Gamma x$.*

For the model construction, we want to eliminate all distinct nominals that are equationally equivalent. This will allow us to construct a term model of the initial branch in which syntactically distinct nominals denote distinct states. This is achieved by considering the image of a branch Γ under a normalizer φ. Of course, applying φ to Γ will destroy the forest structure of G_Γ. The desired properties of $\varphi\Gamma$ can be formulated as follows.

A set Γ of formulas is *saturated for a formula $sx \in \Gamma$ on a set $X \subseteq \mathcal{N}\Gamma$* if $\mathcal{N}(\mathrm{Mod}\,\Gamma) \subseteq X$ and one of the following *saturatedness conditions* holds:

($\mathcal{S}_\wedge$) $s = t_1 \mathbin{\dot\wedge} t_2$ and $t_1x, t_2x \in \Gamma$
($\mathcal{S}_\vee$) $s = t_1 \mathbin{\dot\vee} t_2$ and $t_1x \in \Gamma$ or $t_2x \in \Gamma$
($\mathcal{S}_\Diamond$) $s = [\rho]t$ and either $x \notin X$ or there is some $y \in \mathcal{N}\Gamma$ such that $ty \in \Gamma$, either $\rho xy \in \Gamma$ or $\tilde\rho yx \in \Gamma$, and $\mathcal{L}_\Gamma y = \mathcal{L}_\Gamma z$ for some $z \in X$
($\mathcal{S}_\Box$) $s = [\rho]t$ and, for every y such that $\rho xy \in \Gamma$ or $\tilde\rho yx \in \Gamma$, it holds $ty \in \Gamma$
($\mathcal{S}_E$) $s = Et$ and there is some $y \in X$ such that $ty \in \Gamma$
($\mathcal{S}_A$) $s = At$ and, for every $y \in \mathcal{N}\Gamma$, it holds $ty \in \Gamma$
($\mathcal{S}_N$) $s = \dot{y}$ and $y = x$
($\mathcal{S}_{\bar N}$) $s = \dot\neg\dot{y}$ and $y \dot{\neq} x \in \Gamma$

($\mathcal{S}_@$) $s = @yt$ and $ty \in \Gamma$
($\mathcal{S}_D$) $s = Dt$ and there is some $y \in X$ such that $y \neq x$ and $ty \in \Gamma$
($\mathcal{S}_{\bar{D}}$) $s = \bar{D}t$ and, for every $y \in \mathcal{N}\Gamma$, either $y = x$ or $ty \in \Gamma$

Note that all of the saturatedness conditions but $\mathcal{S}_\Diamond$, $\mathcal{S}_E$, $\mathcal{S}_N$, $\mathcal{S}_D$ and $\mathcal{S}_{\bar{D}}$ are identical to the corresponding expandedness conditions. Γ is called *saturated on a set* $X \subseteq \mathcal{N}\Gamma$ if it is saturated on X for all normal formulas $sx \in \Gamma$. Saturated sets are often also called Hintikka sets after the inventor of the concept.

We define $X_{\Gamma,\varphi} := \{x \in \mathcal{N}(\varphi\Gamma) \mid \exists y \in \mathcal{N}\Gamma \colon y \sim_\Gamma x \text{ and } y \text{ active on } \Gamma\}$. The following proposition captures an essential intuition about $X_{\Gamma,\varphi}$.

Proposition 8. *Let φ be a normalizer for a branch Γ. If x is active on Γ, then $\varphi x \in X_{\Gamma,\varphi}$.*

Proposition 9. *Let φ be a normalizer for a branch Γ. If Γ is open and maximal, then $\varphi\Gamma$ is open and saturated on $X_{\Gamma,\varphi}$.*

Proof. First, we show by contradiction that $\varphi\Gamma$ is open. Assume $\varphi\Gamma$ closed. Then there are some x, y such that $\varphi x = \varphi y$ (which is equivalent to $x \sim_\Gamma y$ since φ is a normalizer) and either $x \dot{\neq} y \in \Gamma$ or $px, \dot{\neg} py \in \Gamma$. In the former case, it immediately follows that Γ is closed, in contradiction to the assumption. In the latter case, the contradiction follows by Lemma 1.

Now to saturatedness on $X_{\Gamma,\varphi}$. Let us first show that $\mathcal{N}(\mathrm{Mod}\,(\varphi\Gamma)) = \varphi(\mathcal{N}(\mathrm{Mod}\,\Gamma)) \subseteq X_{\Gamma,\varphi}$. Let $x \in \mathcal{N}(\mathrm{Mod}\,\Gamma)$. It suffices to show that $\varphi x \in X_{\Gamma,\varphi}$. By the Subterm Property, $x \in \mathcal{N}(\mathrm{Mod}\,\Gamma_0)$, where Γ_0 is the initial branch. Since Γ_0 contains no edges, x is a root of G_{Γ_0}. Then, by Proposition 4, x is a root of G_Γ and hence active on Γ. Since $x \sim_\Gamma \varphi x$, we have $\varphi x \in X_{\Gamma,\varphi}$.

It remains to show that $\varphi\Gamma$ satisfies the respective saturatedness conditions for all normal formulas $sx \in \Gamma$, which we do by case analysis on s. The claim is almost immediate for all cases but $s = \langle\rho\rangle t$, $s = Et$, $s = \dot{y}$, $s = Dt$ and $s = \bar{D}t$, so let us focus on these cases.

Case $s = \dot{y}$ ($\mathcal{S}_N$). It suffices to show that $\varphi y = \varphi x$. By $\mathcal{E}_N$, $y \dot{=} x \in \Gamma$. So, $y \sim_\Gamma x$ and hence $\varphi y = \varphi x$.

Case $s = \bar{D}t$ ($\mathcal{S}_{\bar{D}}$). Similarly to the preceding case.

Case $s = \langle\rho\rangle t$ ($\mathcal{S}_\Diamond$). Let $\varphi x \in X_{\Gamma,\varphi}$. It suffices to show that there is some y such that $(\varphi t)y \in \varphi\Gamma$, $\rho(\varphi x)y \in \Gamma$ or $\tilde{\rho}y(\varphi x) \in \Gamma$, and $\mathcal{L}_{\varphi\Gamma}y = \mathcal{L}_{\varphi\Gamma}z$ for some $z \in X_{\Gamma,\varphi}$.
We know that there is some active u such that $x \sim_\Gamma \varphi x \sim_\Gamma u$. By Lemma 1, $\langle\rho\rangle tu \in \Gamma$. Hence, by $\mathcal{E}_\Diamond$, there is some v such that $tv \in \Gamma$ and either $\rho uv \in \Gamma$ or $\tilde{\rho}vu \in \Gamma$. Since $\varphi u = \varphi x$, $(\varphi t)(\varphi v) \in \varphi\Gamma$ and either $\rho(\varphi x)(\varphi v) \in \varphi\Gamma$ or $\tilde{\rho}(\varphi v)(\varphi x) \in \varphi\Gamma$. So, let $y = \varphi v$. It remains to show that $\mathcal{L}_{\varphi\Gamma}y = \mathcal{L}_{\varphi\Gamma}z$ for some $z \in X_{\Gamma,\varphi}$. Since u is active, by Proposition 6, v is relevant. Hence, by Proposition 7, there is some active w such that $\mathcal{L}_\Gamma v = \mathcal{L}_\Gamma w$. Then, by Lemma 2, $\mathcal{L}_{\varphi\Gamma}y = \mathcal{L}_{\varphi\Gamma}(\varphi w)$. Moreover, by Proposition 8, $\varphi w \in X_{\Gamma,\varphi}$. So, φw is the required z.

Case $s = Dt$ ($\mathcal{S}_D$). It suffices to show that there is some $y \in X_{\Gamma,\varphi}$ such that $y \neq \varphi x$ and $(\varphi t)y \in \varphi\Gamma$.

By $\mathcal{E}_D$, there is some $z \in \mathrm{Root}\,\Gamma$ such that $z \not\sim_\Gamma x$ and $tz \in \Gamma$. Then $(\varphi t)(\varphi z) \in \varphi\Gamma$. As z clearly is active on Γ, by Proposition 8, $\varphi z \in X_{\Gamma,\varphi}$. Moreover, $\varphi z \sim_\Gamma z \not\sim_\Gamma x \sim_\Gamma \varphi x$, i.e. $\varphi z \neq \varphi x$. So, φz is the required y.

Case $s = Et$ ($\mathcal{S}_E$). Analogously to the preceding case, but simpler. □

Given a set Γ saturated on $X \subseteq \mathcal{N}\Gamma$, we call ρxy *safe (for Γ and X)* if $x, y \in X$ and there is some $z \in \mathcal{N}\Gamma$ such that $\rho xz \in \Gamma$ or $\tilde{\rho} zx \in \Gamma$, and $\mathcal{L}_\Gamma z = \mathcal{L}_\Gamma y$. Clearly, if $x, y \in X$ and either $\rho xy \in \Gamma$ or $\tilde{\rho} yx \in \Gamma$, then ρxy is safe. Moreover, ρxy is safe if and only if $\tilde{\rho} yx$ is safe.

Let Γ be saturated on $X \subseteq \mathcal{N}\Gamma$, and let $x_0 \in X$. We define the modal interpretation $\mathfrak{M}^\Gamma$ as follows:

$$\begin{aligned}
\mathfrak{M}^\Gamma \mathrm{S} &= X \\
\mathfrak{M}^\Gamma x &= \text{if } x \in X \text{ then } x \text{ else } x_0 \\
\mathfrak{M}^\Gamma p &= \lambda x \in X.\, \text{if } px \in \Gamma \text{ then } 1 \text{ else } 0 \\
\mathfrak{M}^\Gamma r &= \{(x,y) \mid rxy \text{ safe for } \Gamma \text{ and } X\}
\end{aligned}$$

Proposition 10 (Model Existence). *Let Γ be open and saturated on some $X \subseteq \mathcal{N}\Gamma$. If $x \in X$, s modal and $sx \in \Gamma$, then $\mathfrak{M}^\Gamma$ satisfies sx.*

Proof. By induction on the size of s. The cases $s = p$, $s = \dot{\neg} p$, $s = \dot{y}$, $s = \dot{\neg}\dot{y}$, $s = t_1 \dot{\wedge} t_2$, $s = t_1 \dot{\vee} t_2$, $s = Et$, $s = At$, $s = Dt$ and $s = \bar{D}t$ are easy, so let us focus on the remaining ones.

Case $s = \langle\rho\rangle t$. By $\mathcal{S}_\Diamond$, there is some $y \in \mathcal{N}\Gamma$ such that $ty \in \Gamma$ and either $\rho xy \in \Gamma$ or $\tilde{\rho} yx \in \Gamma$, and some $z \in X$ such that $\mathcal{L}_\Gamma y = \mathcal{L}_\Gamma z$. So, ρxz is safe, i.e. $(x,z) \in \mathfrak{M}^\Gamma \rho$. Moreover, $tz \in \Gamma$. Hence, by the inductive hypothesis, $\mathfrak{M}^\Gamma$ satisfies tz.

Case $s = [\rho]tx$. Let $(x,y) \in \mathfrak{M}^\Gamma \rho$. We have to show that $\mathfrak{M}^\Gamma$ satisfies ty. Clearly, ρxy is safe, so $x, y \in X$ and there is some $z \in \mathcal{N}\Gamma$ such that $\rho xz \in \Gamma$ or $\tilde{\rho} zx \in \Gamma$, and $\mathcal{L}_\Gamma z = \mathcal{L}_\Gamma y$. By $\mathcal{S}_\Box$, it holds $tz \in \Gamma$. Hence $ty \in \Gamma$. The claim follows by the inductive hypothesis. □

Let φ be a substitution and $\mathfrak{M}$ a modal interpretation. We define $\mathfrak{M}_\varphi$ to be the modal interpretation obtained from $\mathfrak{M}$ such that, for all terms s, $\mathfrak{M}_\varphi s = \mathfrak{M}(\varphi s)$.

Proposition 11. *$\mathfrak{M}$ satisfies φs if and only if $\mathfrak{M}_\varphi$ satisfies s.*

Theorem 1 (Model Existence). *Let Γ be open and maximal. Let φ be a normalizer for Γ. If x is active on Γ and $sx \in \Gamma$, then $(\mathfrak{M}^{\varphi\Gamma})_\varphi$ satisfies sx.*

Proof. Let Γ be open and maximal. Let φ be a normalizer for Γ. Let x be active on Γ and $sx \in \Gamma$. By Proposition 9, $\varphi\Gamma$ is open and saturated on $X_{\Gamma,\varphi}$. By Proposition 8, $\varphi x \in X_{\Gamma,\varphi}$. Then, by Proposition 10, $\mathfrak{M}^{\varphi\Gamma}$ satisfies $\varphi(sx) = (\varphi s)(\varphi x) \in \varphi\Gamma$. Hence, by Proposition 11, $(\mathfrak{M}^{\varphi\Gamma})_\varphi$ satisfies sx. □

Since all nominals on the initial branch Γ_0 are roots of G_{Γ_0} and hence active, the interpretation constructed in Theorem 1 from any open and maximal extension of Γ_0 satisfies Γ_0.

7 Explicit Computation of Equational Equivalence

The rule $\mathcal{R}_{\doteq}$, the expandedness conditions $\mathcal{E}_D$ and $\mathcal{E}_{\bar{D}}$, and the closedness criteria for tableau branches take for granted that the equational equivalence relation $\sim_\Gamma$ can be effectively computed. We leave open how this computation is performed. Alternatively, one could make the computation of $\sim_\Gamma$ explicit by replacing $\mathcal{R}_{\doteq}$ by the following two rules.

$$\mathcal{R}_{\doteq}^{\text{sub}} \ \frac{sx \qquad x\doteq y}{sy} \ s \text{ modal} \qquad\qquad \mathcal{R}_{\doteq}^{\text{sym}} \ \frac{x\doteq y}{y\doteq x}$$

Additionally, one could change the closedness criteria and the expandedness conditions for the difference modality to work with an explicit syntactic representation of $\sim_\Gamma$. Note, however, that for the so modified calculus to terminate, one needs to ensure that the computation of $\sim_\Gamma$ is performed before the rule $\mathcal{R}_D$ is applied. One way of doing so is as follows. One takes, in addition to $\mathcal{R}_{\doteq}^{\text{sub}}$ and $\mathcal{R}_{\doteq}^{\text{sym}}$, the following rule.

$$\mathcal{R}_{\doteq}^{\not\doteq} \ \frac{x\not\doteq y \qquad y\doteq z}{x\not\doteq z}$$

One then prioritizes $\mathcal{R}_{\doteq}^{\text{sub}}$, $\mathcal{R}_{\doteq}^{\text{sym}}$ and $\mathcal{R}_{\doteq}^{\not\doteq}$ over $\mathcal{R}_D$ while replacing the conditions "$y \not\sim_\Gamma x$" in $\mathcal{E}_D$ and "$y \sim_\Gamma x$" in $\mathcal{E}_{\bar{D}}$ by "$y\doteq x \notin \Gamma$" and "$y\doteq x \in \Gamma$", respectively, and changing the closedness criterion for disequations from "$x\not\doteq y(\in \Gamma)$ where $x \sim_\Gamma y$" to "$x\not\doteq x \in \Gamma$".

We chose $\mathcal{R}_{\doteq}$ over syntactic rules like $\mathcal{R}_{\doteq}^{\text{sub}}$, $\mathcal{R}_{\doteq}^{\text{sym}}$ and $\mathcal{R}_{\doteq}^{\not\doteq}$ to simplify the presentation and because we didn't want to commit to any particular algorithmic treatment of equational equivalence. Moreover, a practical implementation is likely to use a different, more efficient way of computing $\sim_\Gamma$ than the one suggested by the above rules.

8 Conclusion

We have seen a terminating tableau calculus for basic hybrid logic with converse and difference. Termination of the calculus was obtained by combining chain-based blocking for logics with converse as introduced by Horrocks and Sattler [15] with a complete and terminating treatment of D in [10]. To prove completeness of the calculus, it was necessary to refine conventional filtration arguments as found in [15,14] by distinguishing between modal and equational equivalence of states.

Following [29,15], one can further extend our calculus to cover reflexive, symmetric and transitive modalities while retaining termination. Since the depth of G_Γ is bounded by an exponential in the size of the input, the size of our tableau branches is at most doubly exponential. Hence, a naive implementation would have triply exponential worst-case complexity. Donini and Massacci [30] and later Goré and Nguyen [31] show that caching of satisfiability results for explored tableau branches can reduce the complexity of tableau algorithms for expressive nominal-free description logics to ExpTime, resulting in decision procedures that are worst-case optimal [32,8]. It is an open problem to find corresponding techniques that would scale to logics with nominals and difference.

Acknowledgment. We would like to thank the anonymous referees for their valuable comments and suggestions to this paper.

References

1. Balbiani, P., Demri, S.: Prefixed tableaux systems for modal logics with enriched languages. In: Ralescu, A.L., Shanahan, J.G. (eds.) Proc. 15th Intl. Joint Conf. on Artificial Intelligence (IJCAI 1997), pp. 190–195. Morgan Kaufmann, San Francisco (1997)
2. Blackburn, P., de Rijke, M., Venema, Y.: Modal Logic. Cambridge University Press, Cambridge (2001)
3. Gargov, G., Goranko, V.: Modal logic with names. Journal of Philosophical Logic 22, 607–636 (1993)
4. Areces, C., ten Cate, B.: Hybrid logics. In: [33]
5. Goranko, V.: Modal definability in enriched languages. Notre Dame Journal of Formal Logic 31(1), 81–105 (1990)
6. de Rijke, M.: The modal logic of inequality. J. Symb. Log. 57(2), 566–584 (1992)
7. Venema, Y.: Derivation rules as anti-axioms in modal logic. J. Symb. Log. 58(3), 1003–1034 (1993)
8. Areces, C., Blackburn, P., Marx, M.: The computational complexity of hybrid temporal logics. L. J. of the IGPL 8(5), 653–679 (2000)
9. Fitting, M.: Modal proof theory. In: [33]
10. Kaminski, M., Smolka, G.: Hybrid tableaux for the difference modality. In: 5th Workshop on Methods for Modalities (M4M-5) (2007)
11. Hughes, G.E., Cresswell, M.J.: An Introduction to Modal Logic. Methuen (1968)
12. Halpern, J.Y., Moses, Y.: A guide to completeness and complexity for modal logics of knowledge and belief. Artif. Intell. 54, 319–379 (1992)
13. Horrocks, I., Hustadt, U., Sattler, U., Schmidt, R.: Computational modal logic. In: [33]
14. Bolander, T., Blackburn, P.: Termination for hybrid tableaus. J. Log. Comput. 17(3), 517–554 (2007)
15. Horrocks, I., Sattler, U.: A description logic with transitive and inverse roles and role hierarchies. J. Log. Comput. 9(3), 385–410 (1999)
16. Horrocks, I., Sattler, U., Tobies, S.: Practical reasoning for very expressive description logics. L. J. of the IGPL 8(3), 239–263 (2000)
17. Horrocks, I., Sattler, U.: A tableau decision procedure for $\mathcal{SHOIQ}$. J. Autom. Reasoning 39(3), 249–276 (2007)
18. Hollunder, B., Baader, F.: Qualifying number restrictions in concept languages. In: Allen, J., Fikes, R., Sandewall, E. (eds.) Proc. 2nd Intl. Conf. on Principles of Knowledge Representation and Reasoning (KR 1991), pp. 335–346. Morgan Kaufmann, San Francisco (1991)

19. Calvanese, D., De Giacomo, G., Rosati, R.: A note on encoding inverse roles and functional restrictions in $\mathcal{ALC}$ knowledge bases. In: Franconi, E., De Giacomo, G., MacGregor, R.M., Nutt, W., Welty, C.A. (eds.) Proc. 1998 Intl. Workshop on Description Logics (DL 1998). CEUR Workshop Proceedings, vol. 11, pp. 69–71 (1998)
20. Grädel, E.: On the restraining power of guards. J. Symb. Log. 64(4), 1719–1742 (1999)
21. Ganzinger, H., de Nivelle, H.: A superposition decision procedure for the guarded fragment with equality. In: Proc. 14th Annual IEEE Symposium on Logic in Computer Science (LICS 1999), pp. 295–304. IEEE Computer Society Press, Los Alamitos (1999)
22. Gallin, D.: Intensional and Higher-Order Modal Logic. With Applications to Montague Semantics. Mathematics Studies, vol. 19. North-Holland, Amsterdam (1975)
23. Gamut, L.T.F.: Logic, Language and Meaning. In: Intensional Logic and Logical Grammar, vol. 2. The University of Chicago Press (1991)
24. Carpenter, B.: Type-Logical Semantics. Language, Speech, and Communication. The MIT Press, Cambridge (1997)
25. Hardt, M., Smolka, G.: Higher-order syntax and saturation algorithms for hybrid logic. Electr. Notes Theor. Comput. Sci. 174(6), 15–27 (2007)
26. Benzmüller, C.E., Paulson, L.C.: Exploring properties of normal multimodal logics in simple type theory with LEO-II. In: Benzmüller, C.E., Brown, C.E., Siekmann, J., Statman, R. (eds.) Festschrift in Honor of Peter B. Andrews on His 70th Birthday. Studies in Logic and the Foundations of Mathematics. IFCoLog (to appear)
27. Kaminski, M., Smolka, G.: A straightforward saturation-based decision procedure for hybrid logic. In: Intl. Workshop on Hybrid Logic 2007 (HyLo 2007) (2007)
28. Bolander, T., Braüner, T.: Tableau-based decision procedures for hybrid logic. J. Log. Comput. 16(6), 737–763 (2006)
29. Massacci, F.: Strongly analytic tableaux for normal modal logics. In: Bundy, A. (ed.) CADE 1994. LNCS(LNAI), vol. 814, pp. 723–737. Springer, Heidelberg (1994)
30. Donini, F.M., Massacci, F.: Exptime tableaux for $\mathcal{ALC}$. Artif. Intell. 124(1), 87–138 (2000)
31. Goré, R., Nguyen, L.A.: EXPTIME tableaux with global caching for description logics with transitive roles, inverse roles and role hierarchies. In: Olivetti, N. (ed.) TABLEAUX 2007. LNCS (LNAI), vol. 4548, pp. 133–148. Springer, Heidelberg (2007)
32. Spaan, E.: Complexity of Modal Logics. PhD thesis, ILLC, University of Amsterdam (1993)
33. Blackburn, P., van Benthem, J., Wolter, F. (eds.): Handbook of Modal Logic. Studies in Logic and Practical Reasoning, vol. 3. Elsevier, Amsterdam (2006)

Automata-Based Axiom Pinpointing

Franz Baader[1,⋆] and Rafael Peñaloza[2,⋆⋆]

[1] Theoretical Computer Science, TU Dresden, Germany
baader@tcs.inf.tu-dresden.de
[2] Intelligent Systems, Uni. Leipzig, Germany
penaloza@informatik.uni-leipzig.de

Abstract. Axiom pinpointing has been introduced in description logics (DL) to help the user understand the reasons why consequences hold by computing minimal subsets of the knowledge base that have the consequence in question (MinA). Most of the pinpointing algorithms described in the DL literature are obtained as extensions of tableau-based reasoning algorithms for computing consequences from DL knowledge bases. In this paper, we show that automata-based algorithms for reasoning in DLs can also be extended to pinpointing algorithms. The idea is that the tree automaton constructed by the automata-based approach can be transformed into a weighted tree automaton whose so-called behaviour yields a pinpointing formula, i.e., a monotone Boolean formula whose minimal valuations correspond to the MinAs. We also develop an approach for computing the behaviour of a given weighted tree automaton.

1 Introduction

Description logics (DLs) [2] are a family of logic-based knowledge representation formalisms, which are employed in various application domains, such as natural language processing, configuration, databases, and bio-medical ontologies, but their most notable success so far is the adoption of the DL-based language OWL [15] as standard ontology language for the semantic web. As the size of DL-based ontologies grows, tools that support improving the quality of such ontologies become more important. DL reasoners [14, 13, 22] can be used to detect inconsistencies and to infer other implicit consequences, such as subsumption relationships between concepts or instance relationships between individuals and concepts. However, for a developer or user of a DL-based ontology, it is often quite hard to understand why a certain consequence computed by the reasoner actually follows from the knowledge base. For example, in the current DL version of the medical ontology SNOMED CT[1] the concept *Amputation-of-Finger* is classified as a subconcept of *Amputation-of-Arm*. Finding the six axioms that are responsible for this error (see [9]) among the more than 350,000 terminological axioms of SNOMED without support by an automated reasoning tool is not easy.

⋆ Partially supported by NICTA, Canberra Research Lab.
⋆⋆ Funded by the German Research Foundation (DFG) under grant GRK 446.
[1] see http://www.ihtsdo.org/our-standards/

Axiom pinpointing [19] has been introduced to help developers or users of DL-based ontologies understand the reasons why a certain consequence holds by computing minimal subsets of the knowledge base that have the consequence in question (MinA). Most of the pinpointing algorithms described in the DL literature (e.g., [4, 19, 18, 17, 16]) are obtained as extensions of tableau-based reasoning algorithms [8] for computing consequences from DL knowledge bases. The pinpointing algorithms and proofs of their correctness in these papers are given for a specific DL and a specific type of knowledge base only, and it is not clear to which of the known tableau-based algorithms for DLs the approaches really generalize. For example, the pinpointing extension described in [16], which can deal with general concept inclusions (GCIs) in the DL $\mathcal{ALC}$, follows the approach introduced in [4], but since GCIs require the introduction of so-called blocking conditions into the tableau-based algorithm to ensure termination [8], there are some new non-trivial problems to be solved.

To overcome the problem of having to design a new pinpointing extension for every tableau-based algorithm, we have introduced in [5] a general approach for extending tableau-based algorithms to pinpointing algorithms. This approach has, however, some annoying limitations. First, it only applies to tableau-based algorithms that terminate without requiring any cycle-checking mechanism such as blocking. Second, termination of the tableau-based algorithm one starts with does not necessarily transfer to its pinpointing extension. Though these problems can, in principle, be solved by restricting the general framework to so-called forest tableaux [7, 6], this solution makes the definitions and proofs quite complicated and less intuitive. Also, the approach can still only handle the most simple version of blocking, usually called subset blocking in the DL literature.

In the present paper, we propose a different general approach for obtaining pinpointing algorithms, which also applies to DLs for which the termination of tableau-based algorithms requires the use of appropriate blocking conditions. It is well-known that automata working on infinite trees can often be used to construct worst-case optimal decision procedures for such DLs [10]. In this automata-based approach, the input inference problem Γ is translated into a tree automaton $\mathcal{A}_\Gamma$, which is then tested for emptiness. Basically, our approach transforms the tree automaton $\mathcal{A}_\Gamma$ into a weighted tree automaton working on infinite trees,[2] whose so-called behaviour yields a pinpointing formula, i.e., a monotone Boolean formula that encodes all the MinAs of Γ. To obtain an actual pinpointing algorithm, we must develop an algorithm for computing the behaviour of weighted tree automata working on infinite trees. We will use the DL $\mathcal{SI}$, which extends the basic DL $\mathcal{ALC}$ [20] with transitive and inverse roles, to illustrate our approach. The use of $\mathcal{SI}$ is, on the one hand, motivated by the fact that the presence of inverses in $\mathcal{SI}$ requires tableau-based algorithms to use a blocking condition that is more sophisticated than subset blocking [8]. On the other hand, the extension of their approach to $\mathcal{SI}$ is mentioned as an open problem in [16].

[2] Although weighted automata working on *finite trees* [21] and weighted automata working on *infinite words* [11] have been considered before, this appears to be the first use of weighted automata working on *infinite trees*.

2 Preliminaries

In this section, we first introduce the DL $\mathcal{SI}$, and then recall the relevant definitions regarding pinpointing from [5].

2.1 The Description Logic $\mathcal{SI}$

As mentioned above, $\mathcal{SI}$ extends $\mathcal{ALC}$ with transitive and inverse roles. An example of a role that should be interpreted as transitive is has-descendant, while has-ancestor should be interpreted as the inverse of has-descendant. Instead of employing the usual approach of "hardcoding" inverse and transitive roles into the syntax and semantics of concept descriptions, we allow the use of inverse and transitivity axioms in the knowledge base. This enables us to pinpoint also these kinds of axioms as reasons for certain consequences. Thus, the concept descriptions that we consider are simply $\mathcal{ALC}$ concept descriptions.

Definition 1 ($\mathcal{ALC}$ concept descriptions). *Let N_C be a set of* concept names *and N_R a set of* role names. The set of $\mathcal{ALC}$ concept descriptions *is the smallest set such that*

- *all concept names are $\mathcal{ALC}$ concept descriptions;*
- *if C and D are $\mathcal{ALC}$ concept descriptions, then so are $\neg C$, $C \sqcup D$, and $C \sqcap D$;*
- *if C is a $\mathcal{ALC}$ concept description and $r \in N_R$, then $\exists r.C$ and $\forall r.C$ are $\mathcal{ALC}$ concept descriptions.*

An interpretation *is a pair $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ where the* domain $\Delta^{\mathcal{I}}$ *is a non-empty set and $\cdot^{\mathcal{I}}$ is a function that assigns to every concept name A a set $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ and to every role name r a binary relation $r^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. This function is extended to $\mathcal{ALC}$ concept descriptions as follows:*

- $(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}, (C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}}, (\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$;
- $(\exists r.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \text{there is a } y \in \Delta^{\mathcal{I}} \text{ with } (x,y) \in r^{\mathcal{I}} \text{ and } y \in C^{\mathcal{I}}\}$;
- $(\forall r.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \text{for all } y \in \Delta^{\mathcal{I}}, (x,y) \in r^{\mathcal{I}} \text{ implies } y \in C^{\mathcal{I}}\}$.

In this paper we restrict the attention to terminological knowledge, which is given by a so-called TBox.

Definition 2 ($\mathcal{SI}$ TBoxes). *An $\mathcal{SI}$* TBox *is a finite set of axioms of the following form: (i) $C \sqsubseteq D$ where C and D are $\mathcal{ALC}$ concept descriptions (GCI); (ii)* $\mathsf{trans}(r)$ *where $r \in N_R$ (transitivity axiom); (iii)* $\mathsf{inv}(r,s)$*, where $r \neq s \in N_R$ (inverse axiom), such that every $r \in N_R$ appears in at most one inverse axiom.*

An interpretation $\mathcal{I}$ is called a model *of the $\mathcal{SI}$ TBox $\mathcal{T}$ if it satisfies all axioms in $\mathcal{T}$, i.e., if (i) $C \sqsubseteq D \in \mathcal{T}$ implies $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$; (ii)* $\mathsf{trans}(r) \in \mathcal{T}$ *implies that $r^{\mathcal{I}}$ is transitive; (iii)* $\mathsf{inv}(r,s) \in \mathcal{T}$ *implies that $(x,y) \in r^{\mathcal{I}}$ iff $(y,x) \in s^{\mathcal{I}}$.*

The main inference problems for terminological knowledge are satisfiability and subsumption

Definition 3 (satisfiability, subsumption). *Let C and D be $\mathcal{ALC}$ concept descriptions and $\mathcal{T}$ an $\mathcal{SI}$ TBox. We say that C is* satisfiable w.r.t. $\mathcal{T}$ *if there is a model $\mathcal{I}$ of $\mathcal{T}$ such that $C^{\mathcal{I}} \neq \emptyset$. In this case, $\mathcal{I}$ is also called a* model of C w.r.t. $\mathcal{T}$. *We call C* unsatisfiable w.r.t. $\mathcal{T}$ *if it does not have a model w.r.t. $\mathcal{T}$. Finally, we say that C* is subsumed *by D w.r.t. $\mathcal{T}$ if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ holds in every model $\mathcal{I}$ of $\mathcal{T}$.*

We want to pinpoint reasons for unsatisfiability and for subsumption. Since C is subsumed by D w.r.t. $\mathcal{T}$ iff $C \sqcap \neg D$ is unsatisfiable w.r.t. $\mathcal{T}$, it is obviously sufficient to design a pinpointing algorithm for unsatisfiability.

The automata-based approach for deciding (un)satisfiability uses the fact that an $\mathcal{ALC}$ concept description C is satisfiable w.r.t. an $\mathcal{SI}$ TBox $\mathcal{T}$ iff it has a certain tree-shaped model, called Hintikka tree for C and $\mathcal{T}$. It constructs a tree automaton whose runs are exactly the Hintikka trees for C and $\mathcal{T}$ (see Section 4.2), and then tests this automaton for emptiness.

2.2 Basic Definitions for Pinpointing

Following [5], we define pinpointing not for a specific DL and inference problem, but rather in a more general setting. The type of inference problem that we will consider is deciding a so-called c-property for a given set of axiomatized inputs.

Definition 4 (axiomatized input, c-property). *Let $\mathfrak{I}$ and $\mathfrak{T}$ be sets of* inputs *and* axioms, *respectively, and let $\mathscr{P}_{admis}(\mathfrak{T}) \subseteq \mathscr{P}_{fin}(\mathfrak{T})$ be a set of finite subsets of $\mathfrak{T}$ such that $\mathcal{T} \in \mathscr{P}_{admis}(\mathfrak{T})$ implies $\mathcal{T}' \in \mathscr{P}_{admis}(\mathfrak{T})$ for all $\mathcal{T}' \subseteq \mathcal{T}$. An* axiomatized input *for $\mathfrak{I}$ and $\mathscr{P}_{admis}(\mathfrak{T})$ is of the form $(\mathcal{I}, \mathcal{T})$ where $\mathcal{I} \in \mathfrak{I}$ and $\mathcal{T} \in \mathscr{P}_{admis}(\mathfrak{T})$.*

A consequence property *(or* c-property *for short) is a set $\mathcal{P} \subseteq \mathfrak{I} \times \mathscr{P}_{admis}(\mathfrak{T})$ such that $(\mathcal{I}, \mathcal{T}) \in \mathcal{P}$ implies $(\mathcal{I}, \mathcal{T}') \in \mathcal{P}$ for every $\mathcal{T}' \in \mathscr{P}_{admis}(\mathfrak{T})$ with $\mathcal{T}' \supseteq \mathcal{T}$.*

For example, let $\mathfrak{I}$ consists of all $\mathcal{ALC}$ concept descriptions, $\mathfrak{T}$ of all GCIs, transitivity axioms, and inverse axioms, and $\mathscr{P}_{admis}(\mathfrak{T})$ of all $\mathcal{SI}$ TBoxes. The following is a c-property: $\mathcal{P} = \{(C, \mathcal{T}) \mid C \text{ is unsatisfiable w.r.t. } \mathcal{T}\}$.

Definition 5. *Given an axiomatized input $\Gamma = (\mathcal{I}, \mathcal{T})$ and a c-property $\mathcal{P}$, a set of axioms $\mathcal{S} \subseteq \mathcal{T}$ is called a* minimal axiom set (MinA) *for Γ w.r.t. $\mathcal{P}$ if $(\mathcal{I}, \mathcal{S}) \in \mathcal{P}$ and $(\mathcal{I}, \mathcal{S}') \notin \mathcal{P}$ for every $\mathcal{S}' \subset \mathcal{S}$. The set of all MinAs for Γ w.r.t. $\mathcal{P}$ is denoted by $\mathsf{MIN}_{\mathcal{P}(\Gamma)}$.*

Note that the notion of a MinA is only interesting if $\Gamma \in \mathcal{P}$; otherwise, the monotonicity requirement for $\mathcal{P}$ entails that $\mathsf{MIN}_{\mathcal{P}(\Gamma)} = \emptyset$. In our example, consider the axiomatized input $\Gamma = (A \sqcap \forall r.C, \mathcal{T})$ where $\mathcal{T}$ consists of

$$\mathrm{ax}_1\colon\ A \sqsubseteq \exists r.B, \quad \mathrm{ax}_2\colon\ B \sqsubseteq \forall s.\neg A, \quad \mathrm{ax}_3\colon\ C \sqsubseteq \neg B, \quad \mathrm{ax}_4\colon\ \mathsf{inv}(r,s) \tag{1}$$

It is easy to see that $\Gamma \in \mathcal{P}$, and that the set of all MinAs for Γ is $\mathsf{MIN}_{\mathcal{P}(\Gamma)} = \{\{\mathrm{ax}_1, \mathrm{ax}_2, \mathrm{ax}_4\}, \{\mathrm{ax}_1, \mathrm{ax}_3\}\}$.

Instead of computing all MinAs, one can also compute a pinpointing formula. To define this formula, we assume that every axiom $t \in \mathfrak{T}$ is labeled with a unique propositional variable, $\mathsf{lab}(t)$. Let $\mathsf{lab}(\mathcal{T})$ be the set of all propositional variables labeling an axiom in $\mathcal{T}$. A *monotone Boolean formula* over $\mathsf{lab}(\mathcal{T})$ is a Boolean formula using variables in $\mathsf{lab}(\mathcal{T})$ and only the connectives conjunction and disjunction. In addition, the constants $\top$ and $\bot$, which always evaluate to true and false, respectively, are monotone Boolean formulae. We identify a propositional *valuation* with the set of propositional variables that it makes true. For a valuation $\mathcal{V} \subseteq \mathsf{lab}(\mathcal{T})$, let $\mathcal{T}_\mathcal{V} = \{t \in \mathcal{T} \mid \mathsf{lab}(t) \in \mathcal{V}\}$.

Definition 6 (pinpointing formula). *Given a c-property $\mathcal{P}$ and an axiomatized input $\Gamma = (\mathcal{I}, \mathcal{T})$, the monotone Boolean formula ϕ over $\mathsf{lab}(\mathcal{T})$ is called a* pinpointing formula *for Γ w.r.t. $\mathcal{P}$ if the following holds for every valuation $\mathcal{V} \subseteq \mathsf{lab}(\mathcal{T})$: $(\mathcal{I}, \mathcal{T}_\mathcal{V}) \in \mathcal{P}$ iff $\mathcal{V}$ satisfies ϕ.*

In our example, we can take $\mathsf{lab}(\mathcal{T}) = \{\mathrm{ax}_1, \dots, \mathrm{ax}_4\}$ as set of propositional variables. It is easy to see that $\mathrm{ax}_1 \wedge ((\mathrm{ax}_2 \wedge \mathrm{ax}_4) \vee \mathrm{ax}_3)$ is a pinpointing formula for Γ w.r.t. $\mathcal{P}$.

Valuations can be ordered by set inclusion. The following is an immediate consequence of the definition of a pinpointing formula [4]: if ϕ a pinpointing formula for Γ w.r.t. $\mathcal{P}$, then

$$\mathsf{MIN}_{\mathcal{P}(\Gamma)} = \{\mathcal{T}_\mathcal{V} \mid \mathcal{V} \text{ is a minimal valuation satisfying } \phi\},$$

This shows that it is enough to design an algorithm for computing a pinpointing formula to obtain all MinAs. However, the reduction suggested by the above identity is not polynomial. One possible way to obtain $\mathsf{MIN}_{\mathcal{P}(\Gamma)}$ from ϕ is to first transform ϕ into disjunctive normal form, and then remove superfluous disjuncts. It is well-known that this can cause an exponential blowup. This should, however, not be viewed as a disadvantage of approaches computing the pinpointing formula rather than directly $\mathsf{MIN}_{\mathcal{P}(\Gamma)}$. If such a blowup happens, then the pinpointing formula actually yields a compact representation of all MinAs.

3 Looping Tree Automata

In this section, we introduce both unweighted and weighted looping tree automata. These automata receive infinite trees of a fixed arity k as inputs. For a positive integer k, we denote the set $\{1, \dots, k\}$ by K. The nodes of our trees can be identified by words in K^* in the usual way: the root node is identified by the empty word ε, and the i-th successor of the node u is identified by ui for $1 \le i \le k$. In the case of labeled trees, we will refer to the labeling of the node $u \in K^*$ in the tree r by $r(u)$. We will also use $\overrightarrow{r(u)}$ to denote the tuple $\overrightarrow{r(u)} = (r(u), r(u1), \dots, r(uk))$. An infinite tree r with labels from a set Q can be represented as a mapping $r : K^* \to Q$.

For our purpose, it is sufficient to use unlabeled infinite trees as inputs for our tree automata. For a fixed arity k, there is exactly one such tree, which we can identify with the set of its nodes, i.e., with K^*.

Definition 7 (looping tree automaton). *A* looping tree automaton *for arity k is a tuple (Q, Δ, I), where Q is a finite set of* states*, $\Delta \subseteq Q^{k+1}$ is the* transition relation*, and $I \subseteq Q$ is the set of* initial states*. A* run *of this automaton on the unlabeled tree K^* is a labeled k-ary tree $r : K^* \to Q$ such that $\overrightarrow{r(u)} \in \Delta$ for all $u \in K^*$. This run is* successful *if $r(\varepsilon) \in I$. The* emptiness problem *for looping tree automata for arity k is the problem of deciding whether a given such automaton has a successful run or not.*

The emptiness problem for looping tree automata can be decided using time polynomial in the size of the automaton. The decision procedure computes all the *bad* states, i.e., states that do not occur in any run, in a *bottom-up* fashion [24, 10]: starting with the empty set as initial set of known bad states, the algorithm iteratively adds states to this set as follows: if all the transitions starting from a state q contain a known bad state, then q is also known to be bad. Note that the condition for adding a bad state q also applies to states that have no transition. Obviously, the set of known bad states becomes stable after at most $|Q|$ iterations, and it is easy to show that the set of states obtained this way is indeed the set of all bad states. The automaton has a successful run iff there is an initial state that is *not* bad.

We will later extend automata-based decision procedures into algorithms that compute pinpointing formulae by transforming looping tree automata into weighted looping automata. The weights of such automata come from a complete distributive lattice [12].

Definition 8 (complete distributive lattice). *A* complete distributive lattice *is a partially ordered set $(S, \leq_S)$ such that infima and suprema of arbitrary subsets of S always exist and distribute over each other.*

In the following, we will often simple use the carrier set S to denote the complete distributive lattice $(S, \leq_S)$. The infimum (supremum) of a subset $T \subseteq S$ will be denoted by $\bigotimes_{t \in T} t$ ($\bigoplus_{t \in T} t$). For the infimum (supremum) of two elements, we will also use infix notation, i.e., write $t_1 \otimes t_2$ ($t_1 \oplus t_2$) to denote the infimum (supremum) of the set $\{t_1, t_2\}$. The least element of S (i.e., the infimum of the whole set S) will be denoted by $\mathbf{0}$, and the greatest element (i.e., the supremum of the whole set S) by $\mathbf{1}$.

It should be noted that our assumption that the weights come from a complete distributive lattice is stronger than the one usually encountered in the literature on weighted automata. In fact, for automata working on finite trees, it is sufficient to assume that the weights come from a so-called semiring [21]. In order to have a well-defined behaviour also for weighted automata working on infinite objects, the existence of infinite products and sums is required [11]. The additional properties imposed by our requirement to have a complete distributive lattice (in particular, the idempotency of product and sum) are used to show that we can actually compute the behaviour of our weighted looping automata (see Section 5).

Definition 9 (weighted looping automaton). *Let S be a complete distributive lattice. A* weighted looping automaton (WLA) *on S for arity k is a tuple $\mathcal{A} = (Q, in, wt)$ where Q is a finite set of* states*, $in : Q \rightarrow S$ is the* initial distribution*, and $wt : Q^{k+1} \rightarrow S$ assigns* weights *to transitions. A* run *of $\mathcal{A}$ is a labeled tree $r : K^* \rightarrow Q$. The* weight *of this run is $wt(r) = in(r(\varepsilon)) \otimes \bigotimes_{u \in K^*} wt(\overrightarrow{r(u)})$. The* behaviour *of the automaton $\mathcal{A}$ is $\|\mathcal{A}\| := \bigoplus_{r:K^* \rightarrow Q} wt(r)$.*

For example, the Boolean semiring $\mathbb{B} = (\{0, 1\}, \wedge, \vee, 1, 0)$ is a complete distributive lattice, where the partial order is defined as $1 \leq_{\mathbb{B}} 0$. Note that we have defined 1 to be smaller than 0, and thus conjunction yields the supremum (i.e., is the "addition" $\oplus$) and disjunction yields the infimum (i.e., is the "multiplication" $\otimes$). Likewise, 1 is the the least element $\mathbf{0}$, and 0 is the greatest element $\mathbf{1}$. We can easily transform a given looping tree automaton $\mathcal{A} = (Q, \Delta, I)$ into a WLA $\mathcal{A}_w$ on $\mathbb{B}$ such that the behaviour of $\mathcal{A}_w$ is 0 iff $\mathcal{A}$ has a successful run. In $\mathcal{A}_w$, the initial distribution maps initial states to 0 and all other states to 1; a tuple in Q^{k+1} receives weight 0 if it belongs to Δ, and weight 1 otherwise.

The bottom-up emptiness test for looping tree automata sketched above can be adapted such that it computes the behaviour of $\mathcal{A}_w$ as follows. We construct a function $\mathsf{bad} : Q \rightarrow \{0, 1\}$ such that $\mathsf{bad}(q) = 1$ iff q is a bad state. In the beginning, no state is known to be bad, and thus we start the iteration with $\mathsf{bad}_0(q) = 0$ for all $q \in Q$. Now assume that the function $\mathsf{bad}_i : Q \rightarrow \{0, 1\}$ for $i \geq 0$ has already been computed. In the next step of our iteration, q becomes known to be bad, i.e., $\mathsf{bad}_{i+1}(q)$ is set to 1, if every transition starting from q leads to at least one state already know to be bad. Thus, we have

$$\mathsf{bad}_{i+1}(q) = \bigwedge_{(q,q_1,\ldots,q_k) \in Q^{k+1}} \left(wt(q, q_1, \ldots, q_k) \vee \bigvee_{j=1}^{k} \mathsf{bad}_i(q_j) \right). \tag{2}$$

The function bad is the limit of this iteration, which is reached after at most $|Q|$ iterations. It is easy to see that the behaviour of $\mathcal{A}_w$ is then $\bigwedge_{q \in Q} in(q) \vee \mathsf{bad}(q)$. In Section 5, we will see that the behaviour of a WLA can always be computed by such an iteration.

4 Automata-Based Pinpointing

In this section, we first introduce our general approach for automata-based pinpointing, and then show how it can be applied to finding a pinpointing formula for unsatisfiability in $\mathcal{SI}$.

4.1 The General Approach

Basically, the automata-based approach for deciding a c-property $\mathcal{P}$ takes axiomatized inputs $\Gamma = (\mathcal{I}, \mathcal{T})$ and translates them into looping tree automata $\mathcal{A}_\Gamma$ such that $\Gamma \in \mathcal{P}$ iff $\mathcal{A}_\Gamma$ does *not* have a successful run. For example, the automaton constructed from a concept description C and a TBox $\mathcal{T}$ has a successful run iff C is satisfiable w.r.t. $\mathcal{T}$, where the c-property is *un*satisfiability.

If the translation from Γ to $\mathcal{A}_\Gamma$ is an arbitrary function, then we have no way of knowing how the axioms in $\mathcal{T}$ influence the behaviour of the automaton, and thus it is not clear how to construct a corresponding pinpointing automaton. For this reason, we will assume that the automaton $\mathcal{A}_\Gamma$ for $\Gamma = (\mathcal{I}, \mathcal{T})$ in a certain sense also contains automata for all axiomatized inputs $(\mathcal{I}, \mathcal{T}')$ with $\mathcal{T}' \subseteq \mathcal{T}$, which can be obtained by appropriately restricting the transitions of $\mathcal{A}_\Gamma$. To be more precise, let $\mathcal{A} = (Q, \Delta, I)$ be a looping tree automaton for arity k and $\Gamma = (\mathcal{I}, \mathcal{T})$ an axiomatized input. A function $\mathsf{res} : \mathcal{T} \to \mathscr{P}(Q^{k+1})$ is called a *restricting function.* The restricting function res can be extended to sets of axioms $\mathcal{T}' \subseteq \mathcal{T}$ as follows: $\mathsf{res}(\mathcal{T}') := \bigcap_{t \in \mathcal{T}'} \mathsf{res}(t)$. For $\mathcal{T}' \subseteq \mathcal{T}$, the *$\mathcal{T}'$-restricted subautomaton of $\mathcal{A}$ w.r.t.* res is defined as $\mathcal{A}_{|\mathcal{T}'} := (Q, \Delta \cap \mathsf{res}(\mathcal{T}'), I)$.

Definition 10 (axiomatic automaton). *Let $\mathcal{A} = (Q, \Delta, I)$ be a looping tree automaton for arity k, $\Gamma = (\mathcal{I}, \mathcal{T})$ an axiomatized input, and* $\mathsf{res} : \mathcal{T} \to \mathscr{P}(Q^{k+1})$ *a restricting function. Then we call* $(\mathcal{A}, \mathsf{res})$ *an* axiomatic automaton *for Γ. Given a c-property $\mathcal{P}$, we say that* $(\mathcal{A}, \mathsf{res})$ *is* correct *for Γ w.r.t. $\mathcal{P}$ if the following holds for every $\mathcal{T}' \subseteq \mathcal{T}$: $(\mathcal{I}, \mathcal{T}') \in \mathcal{P}$ iff $\mathcal{A}_{|\mathcal{T}'}$ does not have a successful run.*

Given a correct axiomatic automaton, we show how to transform it into a weighted looping automaton whose behaviour is a pinpointing formula. This weighted automaton uses the $\mathcal{T}$-Boolean semiring, which is defined as $\mathbb{B}^{\mathcal{T}} := (\hat{\mathbb{B}}(\mathcal{T}), \wedge, \vee, \top, \bot)$, where $\hat{\mathbb{B}}(\mathcal{T})$ is the quotient set of all monotone Boolean formulae over $\mathsf{lab}(\mathcal{T})$ by the propositional equivalence relation, i.e., two propositionally equivalent formulae correspond to the same element of $\hat{\mathbb{B}}(\mathcal{T})$. It is easy to see that this semiring is indeed a complete distributive lattice, where the partial order is defined as $\phi \leq \psi$ iff $\psi \to \phi$ is valid. Note that, similar to the case of the Boolean semiring $\mathbb{B}$, conjunction is the semiring addition (i.e., yields the supremum $\oplus$) and disjunction is the semiring multiplication (i.e., yields the infimum $\otimes$). Likewise, $\top$ is the least element $\mathbf{0}$ and $\bot$ is the greatest element $\mathbf{1}$.

Definition 11 (pinpointing automaton). *Let* $(\mathcal{A}, \mathsf{res})$*, with $\mathcal{A} = (Q, \Delta, I)$, be an axiomatic automaton for $\Gamma = (\mathcal{I}, \mathcal{T})$. The* violating function $\mathsf{vio} : Q^{k+1} \to \mathbb{B}^{\mathcal{T}}$ *is given by*

$$\mathsf{vio}(q_0, q_1, \ldots, q_k) = \bigvee_{\{t \in \mathcal{T} \mid (q_0, q_1, \ldots, q_k) \notin \mathsf{res}(t)\}} \mathsf{lab}(t).$$

The pinpointing automaton *induced by* $(\mathcal{A}, \mathsf{res})$ *w.r.t. $\mathcal{T}$ is the WLA* $(\mathcal{A}, \mathsf{res})^{\mathsf{pin}} = (Q, in, wt)$ *on $\mathbb{B}^{\mathcal{T}}$, where*

$$in(q) = \begin{cases} \bot & \textit{if } q \in I, \\ \top & \textit{otherwise;} \end{cases}$$

$$wt(q_0, q_1, \ldots, q_k) = \begin{cases} \mathsf{vio}(q_0, q_1, \ldots, q_k) & \textit{if } (q_0, q_1, \ldots, q_k) \in \Delta, \\ \top & \textit{otherwise.} \end{cases}$$

It is easy to see that, if $r : K^* \to Q$ is a successful run of $\mathcal{A}$, then its weight is given by $wt(r) = \bigvee_{u \in K^*} \mathsf{vio}(\overrightarrow{r(u)})$; otherwise, $wt(r) = \top$. Intuitively, the violating function expresses which axioms are not "satisfied" by a given transition, and thus the weight of a run accumulates all the axioms violated by any of the transitions appearing as labels in it. Removing all the axioms appearing in that formula would yield a subset of axioms which actually allows for this run; and hence, due to correctness, the property does not hold anymore. Conjoining this information for all possible runs leads us to a pinpointing formula.

Theorem 1. *Let $\mathcal{P}$ be a c-property, $\Gamma = (\mathcal{I}, \mathcal{T})$ an axiomatized input, and $(\mathcal{A}, \mathsf{res})$ a correct axiomatic automaton for Γ w.r.t. $\mathcal{P}$. Then $\|(\mathcal{A}, \mathsf{res})^{\mathsf{pin}}\|$ is a pinpointing formula for Γ w.r.t. $\mathcal{P}$.*

Proof. We need to show that, for every valuation $\mathcal{V} \subseteq \mathsf{lab}(\mathcal{T})$, it holds that $\mathcal{V}$ satisfies $\|(\mathcal{A}, \mathsf{res})^{\mathsf{pin}}\|$ iff $(\mathcal{I}, \mathcal{T}_\mathcal{V}) \in \mathcal{P}$. Let $\mathcal{V} \subseteq \mathsf{lab}(\mathcal{T})$. Suppose first that $(\mathcal{I}, \mathcal{T}_\mathcal{V}) \notin \mathcal{P}$. Since $(\mathcal{A}, \mathsf{res})$ is correct for Γ w.r.t. $\mathcal{P}$, there must be a successful run r of $\mathcal{A}_{|\mathcal{T}_\mathcal{V}}$. Consequently, $\overrightarrow{r(u)} \in \mathsf{res}(\mathcal{T}_\mathcal{V})$ holds for every $u \in K^*$, and thus $\mathcal{V}$ cannot satisfy $\mathsf{vio}(\overrightarrow{r(u)})$, for any $u \in K^*$. Since r is a successful run of $\mathcal{A}_{|\mathcal{T}_\mathcal{V}}$, it is also a successful run of $\mathcal{A}$, which implies $wt(r) = \bigvee_{u \in K^*} \mathsf{vio}(\overrightarrow{r(u)})$. Thus, $\mathcal{V}$ does not satisfy $wt(r)$. But then $\mathcal{V}$ also cannot satisfy $\bigwedge_{r:K^* \to Q} wt(r) = \|(\mathcal{A}, \mathsf{res})^{\mathsf{pin}}\|$.

Conversely, if $\mathcal{V}$ does not satisfy $\|(\mathcal{A}, \mathsf{res})^{\mathsf{pin}}\| = \bigwedge_{r:K^* \to Q} wt(r)$, then there must exist a run r such that $\mathcal{V}$ does not satisfy $wt(r)$. This implies that $\overrightarrow{r(u)} \in \mathsf{res}(\mathcal{T}_\mathcal{V})$ for all $u \in K^*$. Consequently, r is a successful run of $\mathcal{A}_{|\mathcal{T}_\mathcal{V}}$, which shows $(\mathcal{I}, \mathcal{T}_\mathcal{V}) \notin \mathcal{P}$. □

4.2 Constructing Axiomatic Automata for $\mathcal{SI}$

If we want to apply Theorem 1 to obtain an automata-based approach for pinpointing unsatisfiability in $\mathcal{SI}$, we must show how, given an $\mathcal{ALC}$ concept description C and an $\mathcal{SI}$ TBox $\mathcal{T}$, we can construct an axiomatic automaton $(\mathcal{A}_{C,\mathcal{T}}, \mathsf{res}_{C,\mathcal{T}})$ that is correct for $(C, \mathcal{T})$ w.r.t. unsatisfiability. For this purpose, we must adapt the known construction of a looping tree automaton for $\mathcal{SI}$ [3] such that it yields an axiomatic automaton.[3]

As mentioned before, the automata-based approach for deciding (un)satisfiability uses the fact that a concept is satisfiable iff it has a so-called Hintikka tree. The automaton to be constructed will have exactly these Hintikka trees as its runs. Intuitively, Hintikka trees are obtained from tree-shaped models by labeling every node with the "relevant" concept descriptions to which it belongs.

As usual, we assume in the following that all concept descriptions are in *negation normal form* (NNF); that is, negation appears only directly in front of concept names. Any $\mathcal{ALC}$ concept description can be transformed into NNF in linear time using de Morgan, duality of quantifiers, and elimination of double negations. We denote the NNF of C by $\mathsf{nnf}(C)$ and $\mathsf{nnf}(\neg C)$ by $\backsim C$. Given an

[3] On the one hand, the construction in [3] is more complex than the one given here since the states of the automata in [3] contain additional information needed for detecting cycles in a run as early as possible, which is not relevant for the present paper. On the other hand, the states of the automata constructed here contain additional information about transitivity needed for defining the restricting function.

$\mathcal{ALC}$ concept description C and an $\mathcal{SI}$ TBox $\mathcal{T}$, the set of relevant concept descriptions is the set of all subdescriptions of C and of the concept descriptions $\backsim D \sqcup E$ for $D \sqsubseteq E \in \mathcal{T}$. We denote this set by $\mathsf{sub}(C, \mathcal{T})$. The set of role names occurring in C or $\mathcal{T}$ is denoted by $\mathsf{rol}(C, \mathcal{T})$. The states of our automaton are so-called Hintikka sets, which in addition to subdescriptions also contain information about which roles are supposed to be transitive.

Definition 12 (Hintikka set). *A set $H \subseteq \mathsf{sub}(C, \mathcal{T}) \cup \mathsf{rol}(C, \mathcal{T})$ is called a* Hintikka set *for $(C, \mathcal{T})$ if the following three conditions are satisfied: (i) if $D \sqcap E \in H$, then $\{D, E\} \subseteq H$; (ii) if $D \sqcup E \in H$, then $\{D, E\} \cap H \neq \emptyset$; and (iii) there is no concept name A such that $\{A, \neg A\} \subseteq H$.*

The Hintikka set H is compatible with the GCI *$D \sqsubseteq E \in \mathcal{T}$ if it is the empty set or contains $\backsim D \sqcup E$. It is* compatible with the transitivity axiom *$\mathsf{trans}(r) \in \mathcal{T}$ if it is the empty set or contains r. Finally, it is* compatible with the inverse axiom *$\mathsf{inv}(r, s) \in \mathcal{T}$, if $r \in H$ implies $s \in H$ and vice versa.*

The arity k of our automaton is determined by the number of existential restrictions, i.e., concept descriptions of the form $\exists r.D$, contained in $\mathsf{sub}(C, \mathcal{T})$. Since we need to know which successor in the tree corresponds to which existential restriction, we fix an arbitrary bijection $\varphi : \{\exists r.D \mid \exists r.D \in \mathsf{sub}(C, \mathcal{T})\} \rightarrow K$. To obtain full k-ary trees, we will use nodes labeled with the empty set (which is a Hintikka set) as dummy nodes. The following Hintikka conditions will be used to define the transitions of our automaton.

Definition 13 (Hintikka condition). *The tuple of Hintikka sets $(H_0, H_1, \ldots, H_k)$ for $(C, \mathcal{T})$ satisfies the* Hintikka condition *if the following holds for every existential restriction $\exists r.D \in \mathsf{sub}(C, \mathcal{T})$: (i) If $\exists r.D \in H_0$, then $H_{\varphi(\exists r.D)}$ contains D as well as every E for which there is a value restriction $\forall r.E \in H_0$; if, in addition, $r \in H_0$, then $\forall r.E$ belongs to $H_{\varphi(\exists r.D)}$ for every value restriction $\forall r.E \in H_0$. (ii) If $\exists r.D \notin H_0$, then $H_{\varphi(\exists r.D)} = \emptyset$.*

This tuple is compatible with the GCI *$D \sqsubseteq E \in \mathcal{T}$ (*compatible with the transitivity axiom *$\mathsf{trans}(r) \in \mathcal{T}$) if all its components are compatible with $D \sqsubseteq E$ ($\mathsf{trans}(r)$). It is* compatible with the inverse axiom *$\mathsf{inv}(r, s) \in \mathcal{T}$ if all its components are compatible with $\mathsf{inv}(r, s)$, and the following holds for all $t \in \{r, s\}$ and $t^- \in \{r, s\} \setminus \{t\}$: for every $\forall t.F \in H_{\varphi(\exists t^-.D)}$, the set H_0 contains F, and additionally $\forall t^-.F$ if $t \in H_0$.*

We are now ready to define the axiomatic automaton for unsatisfiability in $\mathcal{SI}$.

Definition 14 (axiomatic automaton for $\mathcal{SI}$). *Let C be an $\mathcal{ALC}$ concept description, $\mathcal{T}$ an $\mathcal{SI}$ TBox, and k the number of existential restrictions in $\mathsf{sub}(C, \mathcal{T})$. The axiomatic automaton $(\mathcal{A}_{C,\mathcal{T}}, \mathsf{res}_{C,\mathcal{T}})$ has as its first component the looping tree automaton $\mathcal{A}_{C,\mathcal{T}} := (Q, \Delta, I)$, where*

– *Q consists of all Hintikka sets for $(C, \mathcal{T})$;*

- *Δ consists of all $(H_0, H_1, \ldots, H_k) \in Q^{k+1}$ that satisfy the Hintikka condition;*
- *$I := \{H \in Q \mid C \in H\}$.*

The restricting function $\mathsf{res}_{C,\mathcal{T}}$ maps each axiom $t \in \mathcal{T}$ to the set of all tuples in Δ that are compatible with t.

Correctness of this automaton construction can be shown by an adaptation of the arguments used in [3].

Theorem 2. *Let C be an $\mathcal{ALC}$ concept description and $\mathcal{T}$ an $\mathcal{SI}$ TBox. The axiomatic automaton $(\mathcal{A}_{C,\mathcal{T}}, \mathsf{res}_{C,\mathcal{T}})$ is correct for $(C, \mathcal{T})$ w.r.t. unsatisfiability.*

Theorem 1 shows that it is enough to compute the behaviour of the pinpointing automaton $(\mathcal{A}_{C,\mathcal{T}}, \mathsf{res}_{C,\mathcal{T}})^{\mathsf{pin}}$ induced by $(\mathcal{A}_{C,\mathcal{T}}, \mathsf{res}_{C,\mathcal{T}})$ in order to obtain a pinpointing formula for $(C, \mathcal{T})$ w.r.t. unsatisfiability. In the next section, we show how the behaviour of a weighted looping automaton on a complete distributive lattice can be computed.

5 Computing the Behaviour of a WLA

Clearly, the naive approach that directly uses the definition of the behaviour by first computing and then adding up the weights of all runs would not produce a result in finite time since there are infinitely many runs, which are themselves infinite. Instead, we will use a bottom-up method for computing the behaviour, which generalizes the approach sketched in Section 3 for the special case of weighted automata simulating unweighted ones. In the following, we assume that $\mathcal{A} = (Q, in, wt)$ is an arbitrary, but fixed, WLA on the complete distributive lattice S.

We will show that the WLA $\mathcal{A}$ induces a monotone operator $\mathcal{O} : S^Q \to S^Q$, where S^Q is the set of all mappings from Q to S, and that the behaviour of $\mathcal{A}$ can easily be obtained from the greatest fixpoint of this operator. The partial order $\leq_S$ on S can be transferred to S^Q in the usual way, by applying it componentwise: if $\sigma, \sigma' \in S^Q$, then $\sigma \leq_{S^Q} \sigma'$ iff $\sigma(q) \leq_S \sigma'(q)$ for all $q \in Q$. It is easy to see that $(S^Q, \leq_{S^Q})$ is again a complete distributive lattice S. We will use $\otimes$ and $\oplus$ also to denote the infimum and supremum in S^Q. The least (greatest) element of S^Q is the function $\widetilde{\mathbf{0}}$ ($\widetilde{\mathbf{1}}$) that maps every $q \in Q$ to $\mathbf{0}$ ($\mathbf{1}$).

Following the idea of Equation (2), the operator $\mathcal{O}$ is defined as follows for every $\sigma \in S^Q$:

$$\mathcal{O}(\sigma)(q) = \bigoplus_{(q,q_1,\ldots,q_k) \in Q^{k+1}} wt(q, q_1, \ldots, q_k) \otimes \bigotimes_{j=1}^{k} \sigma(q_j). \tag{3}$$

Lemma 1. *The operator $\mathcal{O}$ is continuous, i.e., for any increasing sequence $\sigma_0 \leq_{S^Q} \sigma_1 \leq_{S^Q} \sigma_2 \leq_{S^Q} \ldots$ we have $\bigoplus_{i \geq 0} \mathcal{O}(\sigma_i) = \mathcal{O}(\bigoplus_{i \geq 0} \sigma_i)$.*

Proof. For any sequence $\sigma_0, \sigma_1, \sigma_2, \ldots$ of elements of S^Q we have

$$\begin{aligned}
\bigoplus_{i\geq 0} \mathcal{O}(\sigma_i)(q) &= \bigoplus_{i\geq 0} \bigoplus_{(q,q_1,\ldots,q_k)\in Q^{k+1}} wt(q,q_1,\ldots,q_k) \otimes \bigotimes_{j=1}^{k} \sigma_i(q_j) \\
&= \bigoplus_{(q,q_1,\ldots,q_k)\in Q^{k+1}} \bigoplus_{i\geq 0} wt(q,q_1,\ldots,q_k) \otimes \bigotimes_{j=1}^{k} \sigma_i(q_j) \\
&= \bigoplus_{(q,q_1,\ldots,q_k)\in Q^{k+1}} wt(q,q_1,\ldots,q_k) \otimes \bigoplus_{i\geq 0} \bigotimes_{j=1}^{k} \sigma_i(q_j) \qquad (4) \\
&= \bigoplus_{(q,q_1,\ldots,q_k)\in Q^{k+1}} wt(q,q_1,\ldots,q_k) \otimes \bigotimes_{j=1}^{k} \bigoplus_{i\geq 0} \sigma_i(q_j) \qquad (5) \\
&= \mathcal{O}(\bigoplus_{i\geq 0} \sigma_i),
\end{aligned}$$

where Identities (4) and (5) are a consequence of distributivity. □

By Tarski's fixpoint theorem [23], this implies that $\mathcal{O}$ has $\bigotimes_{n\geq 0} \mathcal{O}^n(\widetilde{\mathbf{1}})$ as its greatest fixpoint (gfp). If S happens to be finite, and hence also S^Q is finite, this gfp is obviously reached after finitely many iterations, i.e., there exists a smallest $m, 0 \leq m \leq |S|^{|Q|}$ such that $\mathcal{O}^m(\widetilde{\mathbf{1}}) = \mathcal{O}^{m+1}(\widetilde{\mathbf{1}})$, and for this m we have $\bigotimes_{n\geq 0} \mathcal{O}^n(\widetilde{\mathbf{1}}) = \mathcal{O}^m(\widetilde{\mathbf{1}})$. Note that the complete distributive lattice $\mathbb{B}^{\mathcal{T}}$ that we have used in our pinpointing automaton is actually finite. Nevertheless, we will not make a finiteness assumption on the complete distributive lattice S when showing that the behaviour of any WLA on S can effectively be computed.

Next, we show how the gfp of $\mathcal{O}$ relates to the behaviour of the given WLA $\mathcal{A}$. Let $K_q^* := \{r : K^* \to Q \mid r(\varepsilon) = q\}$ denote the set of all runs whose root is labeled with q. Consider the function $\sigma^{\|} \in S^Q$ where $\sigma^{\|}(q) := \bigoplus_{r\in K_q^*} \bigotimes_{u\in K^*} wt(\overrightarrow{r(u)})$. Given this function, we can obtain the behaviour of the WLA $\mathcal{A}$ as follows:

Lemma 2. $\|\mathcal{A}\| = \bigoplus_{q\in Q} in(q) \otimes \sigma^{\|}(q)$.

It turns out that $\sigma^{\|}$ is in fact the greatest fixpoint of $\mathcal{O}$. We will prove this with the help of so-called partial runs. For $n \geq 0$ define $K^{\leq n} := \bigcup_{i=0}^{n} K^i$, and $K^{\leq -1} := \emptyset$. A *partial run of depth* m is simply a mapping $r : K^{\leq m} \to Q$. The set of partial runs with root label $q \in Q$ is denoted by $K_q^{\leq m}$.

Lemma 3. *For all* $n \geq 0$ *and* $q \in Q$ *it holds that*

$$\mathcal{O}^n(\widetilde{\mathbf{1}})(q) = \bigoplus_{r\in K_q^{\leq n}} \bigotimes_{u\in K^{\leq n-1}} wt(\overrightarrow{r(u)}).$$

Proof. The proof is by induction on n. For $n = 0$, we have that, on the one hand, $\mathcal{O}^0(\widetilde{\mathbf{1}})(q) = \widetilde{\mathbf{1}}(q) = \mathbf{1}$. On the other hand, since $K^{\leq -1} = \emptyset$, we have

$\bigotimes_{u\in K^{\leq -1}} wt(\overrightarrow{r(u)}) = \mathbf{1}$. Now, assume that the above identity holds for n.

$$\mathcal{O}^{n+1}(\widetilde{\mathbf{1}})(q) = \bigoplus_{(q_1,\dots,q_k)\in Q^k} wt(q,q_1,\dots,q_k) \otimes \bigotimes_{j=1}^{k} \mathcal{O}^n(\widetilde{\mathbf{1}})(q_j) \tag{6}$$

$$= \bigoplus_{(q_1,\dots,q_k)\in Q^k} wt(q,q_1,\dots,q_k) \otimes \bigotimes_{j=1}^{k} \bigoplus_{r_j\in K_{q_j}^{\leq n}} \bigotimes_{u\in K^{\leq n-1}} wt(\overrightarrow{r_j(u)}) \tag{7}$$

$$= \bigoplus_{(q_1,\dots,q_k)\in Q^k} wt(q,q_1,\dots,q_k) \otimes \bigoplus_{r_1\in K_{q_1}^{\leq n},\dots,r_k\in K_{q_k}^{\leq n}} \bigotimes_{j=1}^{k} \bigotimes_{u\in K^{\leq n-1}} wt(\overrightarrow{r_j(u)}) \tag{8}$$

$$= \bigoplus_{(q_1,\dots,q_k)\in Q^k} \bigoplus_{r_1\in K_{q_1}^{\leq n},\dots,r_k\in K_{q_k}^{\leq n}} wt(q,q_1,\dots,q_k) \otimes \bigotimes_{j=1}^{k} \bigotimes_{u\in K^{\leq n-1}} wt(\overrightarrow{r_j(u)}) \tag{9}$$

$$= \bigoplus_{r\in K_q^{\leq n+1}} wt(\overrightarrow{r(\varepsilon)}) \otimes \bigotimes_{j=1}^{k} \bigotimes_{u\in K^{\leq n-1}} wt(\overrightarrow{r(ju)}) \tag{10}$$

$$= \bigoplus_{r\in K_q^{\leq n+1}} wt(\overrightarrow{r(\varepsilon)}) \otimes \bigotimes_{v\in K^{\leq n}\setminus\{\varepsilon\}} wt(\overrightarrow{r(v)})$$

$$= \bigoplus_{r\in K_q^{\leq n+1}} \bigotimes_{u\in K^{\leq n}} wt(\overrightarrow{r(u)})$$

Identity (6) employs the definition of $\mathcal{O}$, and (7) the induction hypothesis. Identity (8) uses the fact that S^Q is a distributive lattice, which allows us to multiply out, and Identity (9) multiplies $wt(q,q_1,\dots,q_k)$ in. Identity (10) simplifies the two sums by constructing a run of larger depth. Instead of considering the transition $(q,q_1,\dots,q_k)$ and then runs of depth n starting with $q_1,\dots,q_k$, we simply take the corresponding run of depth $n+1$ starting at q. This run labels the root with q and the successor node j of the root with q_j. Below j we have the former run starting with q_j. The remaining identities are trivial. This completes the proof of the lemma. □

Theorem 3. *The mapping $\sigma^{\|}$ is the greatest fixpoint of $\mathcal{O}$.*

Proof. By Tarski's fixpoint theorem, we know that the greatest fixpoint of $\mathcal{O}$ is $\bigotimes_{n\geq 0}\mathcal{O}^n(\widetilde{\mathbf{1}})$. From Lemma 3, we then obtain

$$\bigotimes_{n\geq 0}\mathcal{O}^n(\widetilde{\mathbf{1}})(q) = \bigotimes_{n\geq 0}\bigoplus_{r\in K_q^{\leq n}}\bigotimes_{u\in K^{\leq n-1}} wt(\overrightarrow{r(u)}) = \bigotimes_{n\geq 0}\bigoplus_{r\in K_q^{*}}\bigotimes_{u\in K^{\leq n-1}} wt(\overrightarrow{r(u)}) =$$
$$= \bigoplus_{r\in K_q^{*}}\bigotimes_{n\geq 0}\bigotimes_{u\in K^{\leq n-1}} wt(\overrightarrow{r(u)}) = \bigoplus_{r\in K_q^{*}}\bigotimes_{u\in K^{*}} wt(\overrightarrow{r(u)}) = \sigma^{\|}(q) \quad \square$$

This theorem, along with Lemma 2, allows us to compute the behaviour of weighted looping automata in a bottom-up fashion. If S is finite, then it is obvious how to compute the fixpoint $\sigma^{\|}$: we know that there exists a least $m \leq |S|^{|Q|}$ such that $\mathcal{O}^m(\widetilde{\mathbf{1}}) = \mathcal{O}^{m+1}(\widetilde{\mathbf{1}})$, and for this m we have $\mathcal{O}^m(\widetilde{\mathbf{1}}) = \sigma^{\|}$. If S is infinite, then it is not a priori clear that the fixpoint can be reached after finitely many iterations. We will now show that it is actually sufficient to apply the operator only $|Q|+1$ times. Note that this result is also useful for the finite case since the bound $|Q|+1$ greatly improves on the generic bound $|S|^{|Q|}$.

Definition 15 (m-complete). *A WLA $\mathcal{A}$ is m-*complete *if, for every partial run $r : K^{\leq m} \to Q$ of depth m, there is a run $s_r : K^* \to Q$ such that*

$$\bigotimes_{u \in K^{\leq m-1}} wt(\overrightarrow{r(u)}) \leq_{S^Q} \bigotimes_{u \in K^*} wt(\overrightarrow{s_r(u)}).$$

Using the fact that $\otimes$ is idempotent, it is easy to see that every WLA is m-complete for any m greater than the number of states $|Q|$. The proof is basically the same as the one given in [3] for the fact that a looping tree automaton has a run iff it has a partial run of depth $> |Q|$.

Theorem 4. *If $\mathcal{A}$ is an m-complete WLA, then $\|\mathcal{A}\| = \bigoplus_{q \in Q} in(q) \otimes \mathcal{O}^m(\widetilde{\mathbf{1}})(q)$.*

Proof. Since Lemma 2 yields $\|\mathcal{A}\| = \bigoplus_{q \in Q} in(q) \otimes \sigma^{\|}(q)$, it suffices to prove that $\mathcal{O}^m(\widetilde{\mathbf{1}}) = \sigma^{\|}$. Furthermore, Theorem 3 implies that $\sigma^{\|}$ is the infimum of $\{\mathcal{O}^n(\widetilde{\mathbf{1}}) \mid n \geq 0\}$. Thus, it is enough to show that $\mathcal{O}^m(\widetilde{\mathbf{1}}) \leq_{S^Q} \sigma^{\|}$.

By Lemma 3, $\mathcal{O}^m(\widetilde{\mathbf{1}})(q) = \bigoplus_{r \in K_q^{\leq m}} \bigotimes_{u \in K^{\leq m-1}} wt(\overrightarrow{r(u)})$. Since $\mathcal{A}$ is m-complete, we can replace every partial run r appearing in the previous identity by the corresponding run s_r, obtaining a greater element in the semiring:

$$\begin{aligned}\mathcal{O}^m(\widetilde{\mathbf{1}})(q) &\leq_{S^Q} \bigoplus_{r \in K_q^{\leq m}} \bigotimes_{u \in K^*} wt(\overrightarrow{s_r(u)}) \\ &\leq_{S^Q} \bigoplus_{s \in K_q^*} \bigotimes_{u \in K^*} wt(\overrightarrow{s(u)}) = \sigma^{\|}(q).\end{aligned}$$

This completes the proof of the theorem. □

Let us apply this result to the pinpointing automaton for $\mathcal{SI}$ constructed in the previous section. This automaton has exponentially many states in the size n of the input $(C, \mathcal{T})$. Thus, we need exponentially many applications of the operator $\mathcal{O}$. It is also easy to see that the time required by each application of $\mathcal{O}$ is exponential in n.

Corollary 1. *Let C be an $\mathcal{ALC}$ concept description and $\mathcal{T}$ an $\mathcal{SI}$ TBox. The pinpointing formula for $(C, \mathcal{T})$ w.r.t. unsatisfiability can be computed in time exponential in the size of $(C, \mathcal{T})$.*

Since even deciding satisfiability of $\mathcal{ALC}$ concept descriptions w.r.t. $\mathcal{SI}$ TBoxes is known to be ExpTime-hard, this bound is optimal.

6 Conclusions

We have introduced a general framework for extending decision procedures based on the construction of looping tree automata to pinpointing algorithms. This framework can also elegantly deal with DLs for which tableau-based decision procedures require sophisticated blocking conditions, and to which consequently the general approach for extending tableau-based decision procedures to pinpointing algorithms introduced in [5] does not apply.

Our framework is based on the use of weighted automata working on infinite trees, which have until now not been studied. One of the main contributions of this paper is an approach for computing the behaviour of such automata. An interesting topic for future work is to check whether this approach can be adapted such that it also works in cases where the weighted automaton is not explicitly given, but rather computed on-the-fly. Finally, it would also be interesting to know how to compute the behaviour of weighted automata working on infinite trees that use a non-trivial acceptance condition for runs, such as the Büchi condition. This would allow us to extend our framework such that it can deal with DLs that require such acceptance conditions, such as the DL $\mathcal{ALC}_{\mathsf{trans}}$, which allows for transitive closure of roles [1].

References

[1] Baader, F.: Augmenting concept languages by transitive closure of roles: An alternative to terminological cycles. In: Proc. of the 12th Int. Joint Conf. on Artificial Intelligence (IJCAI 1991) (1991)

[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., Hladik, J., Peñaloza, R.: Automata can show PSPACE results for description logics. Information and Computation (to appear, 2008)

[4] Baader, F., Hollunder, B.: Embedding defaults into terminological knowledge representation formalisms. J. of Automated Reasoning 14, 149–180 (1995)

[5] Baader, F., Peñaloza, R.: Axiom pinpointing in general tableaux. In: Olivetti, N. (ed.) TABLEAUX 2007. LNCS (LNAI), vol. 4548, pp. 11–27. Springer, Heidelberg (2007)

[6] Baader, F., Peñaloza, R.: Blocking and pinpointing in forest tableaux. LTCS-Report LTCS-08-02, Chair for Automata Theory, Institute for Theoretical Computer Science, Dresden University of Technology, Germany (2008), http://lat.inf.tu-dresden.de/research/reports.html

[7] Baader, F., Peñaloza, R.: Pinpointing in terminating forest tableaux. LTCS-Report LTCS-08-03, Chair for Automata Theory, Institute for Theoretical Computer Science, Dresden University of Technology, Germany (2008), http://lat.inf.tu-dresden.de/research/reports.html

[8] Baader, F., Sattler, U.: An overview of tableau algorithms for description logics. Studia Logica 69, 5–40 (2001)

[9] Baader, F., Suntisrivaraporn, B.: Debugging SNOMED CT using axiom pinpointing in the description logic $\mathcal{EL}^+$. In: Proceedings of the International Conference on Representing and Sharing Knowledge Using SNOMED (KR-MED 2008), Phoenix, Arizona (2008)

[10] Baader, F., Tobies, S.: The inverse method implements the automata approach for modal satisfiability. In: Goré, R.P., Leitsch, A., Nipkow, T. (eds.) IJCAR 2001. LNCS (LNAI), vol. 2083, pp. 92–106. Springer, Heidelberg (2001)
[11] Droste, M., Rahonis, G.: Weighted automata and weighted logics on infinite words. In: Ibarra, O.H., Dang, Z. (eds.) DLT 2006. LNCS, vol. 4036, pp. 49–58. Springer, Heidelberg (2006)
[12] Grätzer, G.: General Lattice Theory, 2nd edn. Birkhäuser, Basel (1998)
[13] Haarslev, V., Möller, R.: RACER system description. In: Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2001) (2001)
[14] Horrocks, I.: Using an expressive description logic: FaCT or fiction. In: Proc. of the 6th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR 1998), pp. 636–647 (1998)
[15] Horrocks, I., Patel-Schneider, P.F., van Harmelen, F.: From SHIQ and RDF to OWL: The making of a web ontology language. Journal of Web Semantics 1(1), 7–26 (2003)
[16] Lee, K., Meyer, T., Pan, J.Z.: Computing maximally satisfiable terminologies for the description logic $\mathcal{ALC}$ with GCIs. In: Proc. of the 2006 Description Logic Workshop (DL 2006), CEUR Electronic Workshop Proceedings (2006)
[17] Parsia, B., Sirin, E., Kalyanpur, A.: Debugging OWL ontologies. In: Ellis, A., Hagino, T. (eds.) Proc. of the 14th International Conference on World Wide Web (WWW 2005), pp. 633–640. ACM Press, New York (2005)
[18] Schlobach, S.: Diagnosing terminologies. In: Veloso, M.M., Kambhampati, S. (eds.) Proc. of the 20th Nat. Conf. on Artificial Intelligence (AAAI 2005), pp. 670–675. AAAI Press/The MIT Press (2005)
[19] Schlobach, S., Cornet, R.: Non-standard reasoning services for the debugging of description logic terminologies. In: Gottlob, G., Walsh, T. (eds.) Proc. of the 18th Int. Joint Conf. on Artificial Intelligence (IJCAI 2003), Acapulco, Mexico, pp. 355–362. Morgan Kaufmann, Los Altos (2003)
[20] Schmidt-Schauß, M., Smolka, G.: Attributive concept descriptions with complements. Artificial Intelligence 48(1), 1–26 (1991)
[21] Seidl, H.: Finite tree automata with cost functions. Theor. Comput. Sci. 126(1), 113–142 (1994)
[22] Sirin, E., Parsia, B.: Pellet: An OWL DL reasoner. In: Proc. of the 2004 Description Logic Workshop (DL 2004), pp. 212–213 (2004)
[23] Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pacific Journal of Mathematics 5, 285–309 (1955)
[24] Vardi, M.Y., Wolper, P.: Automata-theoretic techniques for modal logics of programs. J. of Computer and System Sciences 32, 183–221 (1986); A preliminary version appeared in Proc. of the 16th ACM SIGACT Symp. on Theory of Computing (STOC 1984)

Individual Reuse in Description Logic Reasoning

Boris Motik and Ian Horrocks

University of Oxford, UK

Abstract. Tableau calculi are the state-of-the-art for reasoning in description logics (DL). Despite recent improvements, tableau-based reasoners still cannot process certain knowledge bases (KBs), mainly because they end up building very large models. To address this, we propose a tableau calculus with *individual reuse*: to satisfy an existential assertion, our calculus nondeterministically tries to reuse individuals from the model generated thus far. We present two *expansion strategies*: one is applicable to the DL $\mathcal{ELOH}$ and gives us a worst-case optimal algorithm, and the other is applicable to the DL $\mathcal{SHOIQ}$. Using this technique, our reasoner can process several KBs that no other reasoner can.

1 Introduction

Description Logics (DLs) [2] are used for conceptual modeling in diverse areas of computer science. This is largely due to the practical support for automated reasoning, which can help users during modeling. Practical DL reasoners are mostly based on tableau calculi [2], which are essentially model building algorithms. Tableau calculi, as well as the underlying computational problems, are of high computational complexity, so various optimizations of the basic algorithm have been developed [2, Chapter 9] and incorporated into reasoners such as FaCT++ [16], Pellet [12], and RACER [8].

DLs are often used in life sciences, and this continuously poses new challenges for DL research. For example, GALEN [13] and FMA [14]—detailed and comprehensive models of human anatomy—have both been translated into DLs, but the resulting knowledge bases cannot be processed using existing DL reasoners. To address this problem we recently proposed a novel DL reasoning algorithm [10, 11] based on *hypertableau* [4]. Unlike the standard tableau calculi, the hypertableau calculus is deterministic if a knowledge base can be translated into a Horn theory; furthermore, it uses *anywhere blocking*, which can significantly reduce the sizes of the generated models. Our HermiT reasoner, which implements the new calculus, was the first one to successfully process the original version of GALEN—that is, the version from ten years ago.

Despite these improvements, many DL knowledge bases are still out of HermiT's reach. The hypertableau calculus minimizes nondeterminism, so the remaining performance problems are due to the construction of very large models. We therefore propose an extension to the hypertableau calculus based on *individual reuse*: to satisfy an existential assertion $(\exists R.C)(s)$, our calculus first tries to *reuse* an individual from the model constructed thus far; if this fails, the calculus then introduces a fresh individual. We focus here on the hypertableau calculus;

however, individual reuse could be applied in standard tableau calculi as well. In Section 3, we discuss the practical rationale behind our approach. In particular, we observe that, due to certain restrictions of DL languages, DL knowledge bases are often underconstrained, thus naturally allowing for individual reuse.

If applied naïvely, individual reuse would yield a highly nondeterministic calculus. We address this problem in two steps. First, we encapsulate the part of the calculus that determines which individuals to reuse in an *expansion strategy*, and we identify general conditions that a strategy must satisfy in order for the algorithm to be sound, complete, and terminating.[1] Second, we present two particular strategies. For knowledge bases expressed in $\mathcal{ELOH}$—a fragment of the tractable DL $\mathcal{EL}$++ [1]—we show that individual reuse can be done *deterministically*, yielding a polynomial algorithm, and we identify a close correspondence between our calculus and the $\mathcal{EL}$++ algorithm from [1]. For knowledge bases expressed in $\mathcal{SHOIQ}$—the DL underlying the Semantic Web ontology language OWL—we present an expansion strategy that generalizes the $\mathcal{ELOH}$ one.

We have extended HermiT with individual reuse and have conducted a preliminary evaluation on practical ontologies, of which several are used in life sciences. Unlike FaCT++ and Pellet, HermiT can classify the BAMS ontology[2] and a particular fragment of FMA. Furthermore, HermiT can solve individual classification tests on a "hard" fragment of the current version of GALEN; however, each test takes about 40 seconds, which makes classifying the knowledge base infeasible given the large number of concepts. The new algorithm also improves HermiT's performance on complex ontologies such as DOLCE. Thus, individual reuse seems to promise significant improvements in practical DL reasoning.

Please refer to [11] for nonessential technical details. All proofs are given in `http://web.comlab.ox.ac.uk/people/boris.motik/pubs/mh08reuse.pdf`.

2 Preliminaries

The description logic $\mathcal{SHOIQ}$ is defined as follows. A *$\mathcal{SHOIQ}$ signature* is a triple $\Sigma = (N_R, N_C, N_I)$ consisting of disjoint sets of *atomic roles* N_R, *atomic concepts* N_C, and *individuals* N_I. The set of *roles* is $N_R \cup \{R^- \mid R \in N_R\}$. For $R \in N_R$, let $\mathsf{Inv}(R) = R^-$ and $\mathsf{Inv}(R^-) = R$. An *RBox* $\mathcal{R}$ is a finite set of *role inclusions* $R \sqsubseteq S$ and *transitivity axioms* $\mathsf{Trans}(R)$ for R and S roles. Let $\sqsubseteq^*_\mathcal{R}$ be the reflexive-transitive closure of $\{R \sqsubseteq S, \mathsf{Inv}(R) \sqsubseteq \mathsf{Inv}(S) \mid R \sqsubseteq S \in \mathcal{R}\}$. A role R is *transitive* in $\mathcal{R}$ if a role S exists such that $S \sqsubseteq^*_\mathcal{R} R$, $R \sqsubseteq^*_\mathcal{R} S$, and either $\mathsf{Trans}(S) \in \mathcal{R}$ or $\mathsf{Trans}(\mathsf{Inv}(S)) \in \mathcal{R}$; furthermore, R is *simple* if no transitive role S exists such that $S \sqsubseteq^*_\mathcal{R} R$. The set of *concepts* is the smallest set containing concepts shown in the left-hand side of Table 1, for $A \in N_C$, $a \in N_I$, C and D concepts, R a role, S a simple role, and n a nonnegative integer; concepts of the form $\{a\}$, $\geq n\, S.C$, and $\leq n\, S.C$ are called *nominals*, *at-least*, and *at-most* concepts, respectively. A *TBox* $\mathcal{T}$ is a finite set of *general concept inclusions*

[1] We use "soundness" and "completeness" as in resolution theorem proving: a sound calculus preserves satisfiability; a complete calculus correctly detects satisfiability.

[2] `http://brancusi.usc.edu/bkms/`

Table 1. Model-Theoretic Semantics of $\mathcal{SHOIQ}$

Semantics of Roles and Concepts	Semantics of Axioms
$\top^I = \triangle^I$ $\quad\bot^I = \emptyset$	$C \sqsubseteq D \quad \Rightarrow C^I \subseteq D^I$
$\{s\}^I = \{s^I\}$ $\quad(\neg C)^I = \triangle^I \setminus C^I$	$R \sqsubseteq S \quad \Rightarrow R^I \subseteq S^I$
$(C \sqcap D)^I = C^I \cap D^I$ $\quad(C \sqcup D)^I = C^I \cup D^I$	$\mathsf{Trans}(R) \Rightarrow (R^I)^+ \subseteq R^I$
$(R^-)^I = \{\langle b, a\rangle \mid \langle a, b\rangle \in R^I\}$	$C(a) \quad \Rightarrow a^I \in C^I$
$(\forall R.C)^I = \{x \mid \forall y : \langle x, y\rangle \in R^I \rightarrow y \in C^I\}$	$R(a, b) \quad \Rightarrow \langle a^I, b^I\rangle \in R^I$
$(\exists R.C)^I = \{x \mid \exists y : \langle x, y\rangle \in R^I \wedge y \in C^I\}$	$a \approx b \quad \Rightarrow a^I = b^I$
$(\leq n\, S.C)^I = \{x \mid \sharp\{y \mid \langle x, y\rangle \in S^I \wedge y \in C^I\} \leq n\}$	$a \not\approx b \quad \Rightarrow a^I \neq b^I$
$(\geq n\, S.C)^I = \{x \mid \sharp\{y \mid \langle x, y\rangle \in S^I \wedge y \in C^I\} \geq n\}$	

Note: $\sharp N$ is the number of elements in N, and R^+ is the transitive closure of R.

(GCIs) $C \sqsubseteq D$ for C and D concepts. An *ABox* $\mathcal{A}$ is a finite set of assertions of the form $C(a)$, $R(a, b)$, and (in)equalities $a \approx b$ and $a \not\approx b$, for C a concept, R a role, and a and b individuals. A $\mathcal{SHOIQ}$ *knowledge base* $\mathcal{K}$ is a triple $(\mathcal{R}, \mathcal{T}, \mathcal{A})$.

An *interpretation* for $\mathcal{K}$ is a tuple $I = (\triangle^I, \cdot^I)$, where $\triangle^I$ is a nonempty set, and $\cdot^I$ assigns an element $a^I \in \triangle^I$ to each individual a, a set $A^I \subseteq \triangle^I$ to each atomic concept A, and a relation $R^I \subseteq \triangle^I \times \triangle^I$ to each atomic role R. The function $\cdot^I$ is extended to concepts and roles as shown in the left-hand side of Table 1. I is a *model* of $\mathcal{K}$, written $I \models \mathcal{K}$, if it satisfies all axioms of $\mathcal{K}$ as shown in the right-hand side of Table 1. The basic inference problem for $\mathcal{SHOIQ}$ is checking *satisfiability* of $\mathcal{K}$—that is, checking whether a model of $\mathcal{K}$ exists.

The DL $\mathcal{ELOH}$ [3] is obtained from $\mathcal{SHOIQ}$ by disallowing transitive and inverse roles, and by allowing only concepts of the form $\top$, $\bot$, A, $\{a\}$, $\exists R.C$, and $C_1 \sqcap C_2$. The DL $\mathcal{SHIQ}$ is obtained from $\mathcal{SHOIQ}$ by disallowing nominals.

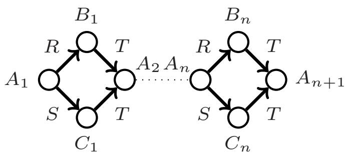
3 Motivation

For a TBox $\mathcal{T}$ and an ABox $\mathcal{A}$, a tableau calculus evolves $\mathcal{A}$ towards a representation of a model by applying derivation rules. The $\exists$-rule is found in virtually all tableau calculi: given $(\exists R.C)(s) \in \mathcal{A}_i$, it introduces a fresh individual t and derives the ABox $\mathcal{A}_{i+1} := \mathcal{A}_i \cup \{R(s, t), C(t)\}$. The rule is applied only if s is not *blocked* in $\mathcal{A}_i$; roughly speaking, this is the case if no individual u exists such that $D(s) \in \mathcal{A}_i$ if and only if $D(u) \in \mathcal{A}_i$ for each concept D. Apart from ensuring termination, blocking significantly reduces model sizes in practice [10].

DLs usually enjoy a *tree model property*: each satisfiable knowledge base has a tree-shaped model. While this is useful for ensuring decidability [17], it also prevents us from fully axiomatizing nontree structures. For example, a structure such as the one shown in Figure 1a can only be approximated using the axioms shown in Figure 1b. The axioms in Figure 1b have a model I corresponding to Figure 1a; however, they also have an exponentially larger model I' obtained by "unfolding" Figure 1a into a tree.

Consider now a run of our hypertableau algorithm on a knowledge base $\mathcal{K}$ containing the axioms in Figure 1b and the axioms $A_{n+1} \sqsubseteq E$, $\exists T.E \sqsubseteq E$, and

(a) The Intended Structure



(b) The Axiomatization in DLs

$$\begin{array}{lcl} A_1 \sqsubseteq \exists R.B_1 \sqcap \exists S.C_1 & & A_n \sqsubseteq \exists R.B_n \sqcap \exists S.C_n \\ B_1 \sqsubseteq \exists T.A_2 & \dots & B_n \sqsubseteq \exists T.A_{n+1} \\ C_1 \sqsubseteq \exists T.A_2 & & C_n \sqsubseteq \exists T.A_{n+1} \end{array}$$

Fig. 1. Problems with Model Construction

$\exists R.E \sqcap \exists S.E \sqsubseteq E$. The algorithm exploits the tree model property by constructing only (representations of) tree-shaped models. It will thus initially construct a fragment of the exponential tree: the fragment will, for example, contain two individuals labeled with A_2 such that one blocks the other. But then E will be added to all existing individuals, which will invalidate blocking. Thus, our algorithm eventually constructs the entire exponential tree. Different blocking and rule application strategies are known that might prevent the generation of the exponential tree in this example; however, the example can be modified such that the exponential tree is generated in spite of these optimizations.

Complex structures abound in life science ontologies; for example, GALEN states that "the left ventricle is a solid division of the left side of the heart," "the left ventricle is a beta connection of the mitral valve," "the mitral valve is a structural component of the left side of the heart," and so on. The intended structure is much more complex than the one shown Figure 1a, and the axioms are cyclic, so tableau reasoners generate very large tree models. It is reasonable, however, to expect that GALEN is satisfied in a relatively small non-tree-shaped model that contains only one left side, one left ventricle, and one mitral valve.

We use these observations in our new calculus: given $(\exists R.C)(s)$, instead of always creating a fresh individual, our calculus first tries to reuse an individual from the model generated thus far. Our calculus is similar to the tableau calculus for first-order logic from [5]; however, the latter calculus is not a decision procedure for DLs that lack the finite-model property and is unlikely to be suitable for practice due to its very large degree of nondeterminism.

4 The Hypertableau Algorithm with Individual Reuse

Our calculus with individual reuse is based on the hypertableau calculi for $\mathcal{SHIQ}$ [10] and $\mathcal{SHOIQ}$ [11]. Therefore, we first present an informal overview of these calculi in Section 4.1, and then formally introduce the calculus with individual reuse in Section 4.2.

4.1 The Standard Hypertableau Calculus for $\mathcal{SHOIQ}$

Our hypertableau calculus is related to hyperresolution with splitting, which has been used to obtain a decision procedure for several description and modal logics [9, 6]. The algorithm consists of the preprocessing and the hypertableau phases.

The *preprocessing phase* translates a $\mathcal{SHOIQ}$ knowledge base $\mathcal{K}$ into a *normalized* ABox $\Xi_{\mathcal{A}}(\mathcal{A})$ and a set $\Xi_{\mathcal{TR}}(\mathcal{K})$ of *DL-clauses*—universally quantified first-order implications of the form $\bigwedge U_i \rightarrow \bigvee V_j$ where U_i and V_j are *atoms* of the form $R(x, y)$, $C(x)$, and $x \approx y$, for x and y variables, R an atomic role, C a concept, and $\approx$ the equality predicate. This transformation is an optimized version of the well-known structural transformation followed by a translation of certain concepts into first-order logic; please refer to [11, Section 4.1] for details. The translation produces *HT-clauses*—syntactically restricted DL-clauses on which our hypertableau calculus is guaranteed to terminate. A precise definition of HT-clauses is given in [11, Definition 7]. Roughly speaking, HT-clauses can have the form (1), where R_i and S_i are (not necessarily atomic) roles, A_i and B_i are atomic concepts, and C_i and D_i are either atomic or concepts of the form $\geq n\, R.A$ or $\geq n\, R.\neg A$. Furthermore, ar is a function defined as $\mathsf{ar}(R, s, t) = R(s, t)$ and $\mathsf{ar}(R^-, s, t) = R(t, s)$ for R an atomic role and s and t individuals or variables.

$$\begin{aligned}(1)\quad & \bigwedge A_i(x) \wedge \bigwedge \mathsf{ar}(R_i, x, y_i) \wedge \bigwedge B_i(y_i) \wedge \bigwedge O_{a_i}(y_{a_i}) \rightarrow \\ & \qquad \bigvee C_i(x) \vee \bigvee D_i(y_i) \vee \bigvee \mathsf{ar}(S_i, x, y_i) \vee \bigvee x \approx y_{a_i} \vee \bigvee y_i \approx y_j\ @^{x}_{\leq n\, R.C}\end{aligned}$$

The atoms of the form $x \approx y_{a_i}$ stem from nominals; for example, $C \sqsubseteq \{a\}$ is translated into an HT-clause $C(x) \wedge O_a(y_a) \rightarrow x \approx y_a$ and an assertion $O_a(a)$. A concept O_a is uniquely associated with each nominal $\{a\}$ and it is called a *nominal guard concept*; such concepts are used to "push" all individuals from DL-clauses into the ABox. Finally, the *at-most equalities* $y_i \approx y_j\ @^{x}_{\leq n\, R.C}$ stem from the translation of at-most concepts; for example, $\top \sqsubseteq\ \leq 1\, R.\top$ is translated into $R(x, y_1) \wedge R(x, y_2) \rightarrow y_1 \approx y_2\ @^{x}_{\leq 1\, R.\top}$. The *annotation* $@^{x}_{\leq 1\, R.\top}$ does not affect the meaning of the equality; it merely records its provenance, and we shall discuss the usage of this provenance information shortly. The concept $\exists R.C$ is used in the rest of this paper as an abbreviation for $\geq 1\, R.C$.

The *hypertableau phase* decides satisfiability of a set of HT-clauses $\mathcal{C}$ and an ABox $\mathcal{A}$. The main derivation rule is similar to the one of the hypertableau calculus for first order logic [4]: given an HT-clause $\bigwedge_{i=1}^{m} U_i \rightarrow \bigvee_{j=1}^{n} V_j$ and an ABox $\mathcal{A}$, the *Hyp*-rule tries to unify the atoms $U_1, \ldots, U_m$ with a subset of the assertions in $\mathcal{A}$; if a unifier σ is found, the rule nondeterministically derives $\sigma(V_j)$ for some $1 \leq j \leq n$. For example, given $R(x, y) \rightarrow \exists R.C(x) \vee D(y)$ and an assertion $R(a, b)$, the *Hyp*-rule derives either $\exists R.C(a)$ or $D(b)$. The $\geq$-rule deals with existential quantifiers: given $\exists R.C(a)$, the rule introduces a fresh individual t and derives $R(a, t)$ and $C(t)$. The $\approx$-rule deals with equality: given $a \approx b$, the rule replaces the individual a in all assertions with the individual b, and it introduces a *renaming* $a \mapsto b$ in order to keep track of the merging. We take $\approx$ to have built-in symmetry; thus, $a \approx b$ should also be read as $b \approx a$. Finally, the $\bot$-rule detects contradictions such as $A(a)$ and $\neg A(a)$, or $a \not\approx a$.

Termination of the calculus is ensured through *blocking*, which is based on the key concept of *forest-shaped ABoxes*. We discuss here just the intuition behind the concept; please refer to [11, Definition 11] for details. A forest-shaped ABox is shown in Figure 2, where nodes and edges correspond to individuals and role assertions, respectively. Individuals in such an ABox can be separated into two

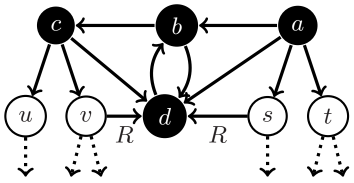


Fig. 2. Forest-Shaped ABoxes

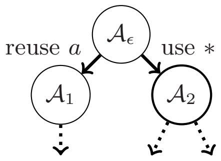


Fig. 3. Example Derivation

sets. *Named* individuals (shown as black nodes) originate from the *input ABox*, and they can be connected in arbitrary ways. *Blockable* individuals (shown as white nodes) are introduced by the $\geq$-rule, and they can be connected either to arbitrary named individuals, or to other blockable individuals in a *tree-like* way. Blockable individuals are represented as strings; for example, $s = a.1$ denotes that s is the first *successor* of a. These notions can be used to identify certain individuals as *blocked*. The $\geq$-rule is applied only to nonblocked individuals, and this ensures termination without affecting completeness.

As shown in [11, Lemma 12], applications of most derivation rules preserve the forest shape of an ABox; however, inverse roles, nominals, and number restrictions cause subtle problems. Consider again Figure 2 and assume that d must satisfy an at-most restriction $\leq 1\, R^-.\top$. This implies $v \approx s$, so one individual should be merged into the other; however, this can compromise the tree shape of the ABox. The *NI*-rule deals with this problem by promoting one of v or s into a *root individual*: such individuals can be connected in arbitrary ways even if they do not occur in the input ABox. Thus, an application of the *Hyp*-rule to the ABox in Figure 2 and the HT-clause $R(y_1, x) \wedge R(y_2, x) \rightarrow y_1 \approx y_2 \,@^x_{\leq 1\, R^-.\top}$ derives the at-most equality $v \approx s \,@^d_{\leq 1\, R^-.\top}$. By examining the annotation on the equality, the *NI*-rule can see that the equality stems from an at-most concept, so it turns either v or s into a root individual. It is possible to establish a bound on the number of the introduced root individuals and thus ensure termination.

4.2 Introducing Individual Reuse

In this section, we extend the hypertableau calculus with individual reuse.

Definition 1 (Hypertableau with Individual Reuse)

Individuals. *Given a set of* named *individuals N_I, the set of* root individuals *N_O is the smallest set such that $N_I \subseteq N_O$ and, if $x \in N_O$, then $x.\langle R, B, i\rangle \in N_O$ for each role R, each integer i, and each B of the form A or $\neg A$ with A an atomic concept. The set of* all individuals *N_A is the smallest set such that $N_O \subseteq N_A$ and, if $x \in N_A$, then $x.i \in N_A$ for each integer i. The individuals in $N_A \setminus N_O$ are* blockable individuals*. A blockable individual $x.i$ is a* successor *of x, and x is a* predecessor *of $x.i$.* Descendant *and* ancestor *are the transitive closures of successor and predecessor, respectively.*

ABoxes. *An ABox that contains only named individuals and no at-most equalities is called an* input ABox*. The hypertableau algorithm works with* generalized ABoxes*, which can contain assertions using the individuals from N_A, a special assertion $\bot$ that is false in all interpretations, and an acyclic and confluent relation $\mapsto$ on root individuals called* renaming*. The* canonical name *of a root individual $a \in N_O$ w.r.t. $\mathcal{A}$, written $\|a\|_\mathcal{A}$, is the normal form of a w.r.t. $\mapsto$ in $\mathcal{A}$. If a occurs in $\mathcal{A}$, then the relation $\mapsto$ must be such that $\|a\|_\mathcal{A} = a$.*

Pairwise Anywhere Blocking. *The* label of an individual *is defined as $\mathcal{L}_\mathcal{A}(s) = \{C \mid C(s) \in \mathcal{A}$ and C is of the form $A, \geq n\, R.A$ or $\geq n\, R.\neg A\}$, and* of an individual pair *as $\mathcal{L}_\mathcal{A}(s,t) = \{R \mid R(s,t) \in \mathcal{A}\}$. Let $\prec$ be a transitive and irreflexive relation on N_A such that, if s' is an ancestor of s, then $s' \prec s$. By induction on $\prec$, we assign to each individual in $\mathcal{A}$ a status as follows: a blockable individual s with a predecessor s' is* directly blocked by *a blockable individual t with a predecessor t' iff t is not blocked, $t \prec s$, $\mathcal{L}_\mathcal{A}(s) = \mathcal{L}_\mathcal{A}(t)$, $\mathcal{L}_\mathcal{A}(s') = \mathcal{L}_\mathcal{A}(t')$, $\mathcal{L}_\mathcal{A}(s,s') = \mathcal{L}_\mathcal{A}(t,t')$, and $\mathcal{L}_\mathcal{A}(s',s) = \mathcal{L}_\mathcal{A}(t',t)$; s is* indirectly blocked *iff it has a predecessor that is blocked; and s is* blocked *iff it is directly or indirectly blocked.*

Pruning. *The ABox $\mathsf{prune}_\mathcal{A}(s)$ is obtained from $\mathcal{A}$ by removing all assertions containing a descendent of s.*

Merging. *The ABox $\mathsf{merge}_\mathcal{A}(s \to t)$ is obtained from $\mathsf{prune}_\mathcal{A}(s)$ by replacing the individual s with the individual t in all assertions (but not in the renaming relation $\mapsto$) and, if both s and t are root individuals, adding the renaming $s \mapsto t$.*

Expansion Strategy. *An* expansion strategy *is a function that, for each concept $\geq n\, R.C$, individual s, and ABox $\mathcal{A}$, returns a k-tuple of n-tuples of the form $\mathsf{iexp}(\geq n\, R.C, s, \mathcal{A}) = [(\gamma_{1,1}, \ldots, \gamma_{1,n}), \ldots, (\gamma_{k,1}, \ldots, \gamma_{k,n})]$, where $k \geq 1$ and, for each $1 \leq i \leq k$ and $1 \leq j \leq n$, $\gamma_{i,j}$ is a special symbol $*$, a predecessor or a successor of s, or a root individual (which may or may not occur in $\mathcal{A}$).*

Derivation Rules. *Table 2 specifies rules that, for $\mathcal{A}$ an ABox, $\mathcal{C}$ a set of HT-clauses, and iexp an expansion strategy, derive the ABoxes $\mathcal{A}_1, \ldots, \mathcal{A}_\ell$.*

Rule Precedence. *The $\approx$-rule can be applied to a (possibly annotated) equality $s \approx t$ in an ABox $\mathcal{A}$ only if $\mathcal{A}$ does not contain an equality of the form $s \approx t\, @^u_{\leq n\, R.B}$ to which the NI-rule is applicable.*

Clash. *An ABox $\mathcal{A}$ contains a* clash *iff $\bot \in \mathcal{A}$; otherwise, $\mathcal{A}$ is* clash-free.

Derivation. *A* derivation *$D = (T, \lambda)$ for a set of HT-clauses $\mathcal{C}$, an ABox $\mathcal{A}$, and an expansion strategy iexp consists of a finitely branching tree T and a function λ labeling the nodes of T with ABoxes such that (*i*) $\lambda(\epsilon) = \mathcal{A}$ for ϵ the root of T, (*ii*) $t \in T$ is a leaf of T if $\bot \in \lambda(t)$ or no derivation rule is applicable to $\lambda(t)$ and $\mathcal{C}$, and (*iii*) otherwise, $t \in T$ has children $t_1, \ldots, t_\ell$ such that $\lambda(t_1), \ldots, \lambda(t_\ell)$ are exactly the results of applying one (arbitrarily chosen, but respecting the precedence) applicable derivation rule to $\lambda(t)$ and $\mathcal{C}$. The derivation D is* successful *if it contains a leaf node labeled with a clash-free ABox.*

The main difference to the hypertableau calculus from [11, Definition 10] is in the $\geq$-rule and the notion of an *expansion strategy*. Intuitively, given an assertion $\geq n\, R.C(s) \in \mathcal{A}$, the new $\geq$-rule consults the expansion strategy iexp to determine the possible ways of satisfying the assertion. The strategy can identify k different ways to do this; if $k > 1$, the $\geq$-rule becomes nondeterministic. The

Table 2. Derivation Rules of the Tableau Calculus

Hyp-rule	If 1. $U_1 \wedge \ldots \wedge U_m \rightarrow V_1 \vee \ldots \vee V_n \in \mathcal{C}$, and 2. a mapping σ of variables to the individuals of $\mathcal{A}$ exists such that 2.1 $\sigma(x)$ is not indirectly blocked for each variable $x \in N_V$, 2.2 $\sigma(U_i) \in \mathcal{A}$ for each $1 \leq i \leq m$, and 2.3 $\sigma(V_j) \notin \mathcal{A}$ for each $1 \leq j \leq n$, then $\mathcal{A}_1 := \mathcal{A} \cup \{\bot\}$ if $n = 0$; $\mathcal{A}_j := \mathcal{A} \cup \{\sigma(V_j)\}$ for $1 \leq j \leq n$ otherwise.
$\geq$-rule	If 1. $\geq n\, R.C(s) \in \mathcal{A}$, 2. s is not blocked in $\mathcal{A}$, 3. $\mathcal{A}$ does not contain individuals $u_1, \ldots, u_n$ such that 3.1 $\{\mathsf{ar}(R, s, u_i), C(u_i) \mid 1 \leq i \leq n\} \cup \{u_i \not\approx u_j \mid 1 \leq i < j \leq n\} \subseteq \mathcal{A}$, and 3.2 either s is blockable or no u_i, $1 \leq i \leq n$, is indirectly blocked in $\mathcal{A}$, and 4. $\mathsf{iexp}(\geq n\, R.C, s, \mathcal{A}) = [(\gamma_{1,1}, \ldots, \gamma_{1,n}), \ldots, (\gamma_{k,1}, \ldots, \gamma_{k,n})]$ then for each $1 \leq i \leq k$, $\mathcal{A}_i := \mathcal{A} \cup \{\mathsf{ar}(R, s, t_{i,j}), C(t_{i,j}) \mid 1 \leq j \leq n\} \cup \{t_{i,j} \not\approx t_{i,k} \mid 1 \leq j < k \leq n\}$ where $t_{i,j}$ is a fresh successor of s if $\gamma_{i,j} = *$, and $t_{i,j} = \gamma_{i,j}$ if $\gamma_{i,j} \neq *$.
$\approx$-rule	If 1. $s \approx t \in \mathcal{A}$ (the equality can possibly be annotated), and 2. $s \neq t$ then $\mathcal{A}_1 := \mathsf{merge}_{\mathcal{A}}(s \rightarrow t)$ if t is a named individual, or t is a root individual and s is not a named individual, or s is a descendant of t; $\mathcal{A}_1 := \mathsf{merge}_{\mathcal{A}}(t \rightarrow s)$ otherwise.
$\bot$-rule	If $s \not\approx s \in \mathcal{A}$ or $\{A(s), \neg A(s)\} \subseteq \mathcal{A}$ then $\mathcal{A}_1 := \mathcal{A} \cup \{\bot\}$.
NI-rule	If 1. $s \approx t\, @^u_{\leq n\, R.B} \in \mathcal{A}$ or $t \approx s\, @^u_{\leq n\, R.B} \in \mathcal{A}$, 2. u is a root individual, 3. s is a blockable individual and it is not a successor of u, and 4. t is a blockable individual then $\mathcal{A}_i := \mathsf{merge}_{\mathcal{A}}(s \rightarrow \lVert u.\langle R, B, i\rangle \rVert_{\mathcal{A}})$ for each $1 \leq i \leq n$.

i-th variant is described as an n-tuple $(\gamma_{i,1}, \ldots, \gamma_{i,n})$, in which $\gamma_{i,j}$ specifies how to obtain the j-th of the n required individuals. For each $\gamma_{i,j}$, the strategy can cause the $\geq$-rule to either (*i*) reuse an existing individual from $\mathcal{A}$, (*ii*) introduce a fresh root individual, or (*iii*) resort to the standard tableau behavior and introduce a fresh distinct blockable successor of s (when $\gamma_{i,j} = *$).

To preserve the forest shape of ABoxes, iexp cannot reuse an arbitrary individual from $\mathcal{A}$. For example, if the strategy reused u when expanding $\exists R.C(s)$ in the ABox shown in Figure 2, the resulting ABox would not be tree-shaped. Therefore, if $\gamma_{i,j} \neq *$, then $\gamma_{i,j}$ must be either a root individual, a predecessor of s, or a successor of s, thus ensuring that the resulting ABox is tree-shaped. A typical expansion strategy will use this definition as follows: the strategy will first introduce fresh root individuals and designate them as possible "reuse targets"; in subsequent inferences, the strategy will try to reuse these targets; finally, if such reuse fails, the strategy will resort to the standard behavior.

A strategy can thus introduce an unbounded number of root individuals; since these do not participate in blocking, the calculus is not guaranteed to terminate.

Therefore, we require a strategy to be *bounded*—that is, to introduce a finite number of fresh individuals in each possible derivation.

Definition 2 (Bounded Strategy). *A strategy* iexp *is* bounded *if the number of individuals* $\gamma_{i,j}$ *such that* $\gamma_{i,j}$ *occurs in* $\mathsf{iexp}(\geq n\, R.C, s, \mathcal{A})$ *but not in* $\mathcal{A}$ *for some concept* $\geq n\, R.C$*, individual* $s \in N_O$*, and ABox* $\mathcal{A}$ *is finite.*

Since individual reuse preserves the forest shape of ABoxes, given a clash-free ABox $\mathcal{A}$ to which no derivation rule is applicable, we can construct a model for $\mathcal{A}$ just like in [11, Lemma 14]. In contrast, individual reuse is *unsound*: an application of the $\geq$-rule to a satisfiable ABox can result in an unsatisfiable ABox. The following definition introduces derivations in which reusing individuals does not lead to unsoundness.

Definition 3 (Safe Derivation). *Let* $D = (T, \lambda)$ *be a derivation for* $\mathcal{C}$*,* $\mathcal{A}$*, and* iexp*. The set of* safe *nodes of* T *is inductively defined such that* $\epsilon \in T$ *is safe, and* $t.i \in T$ *is safe if* t *is safe and (*i*)* $\lambda(t.i)$ *has not been obtained from* $\lambda(t)$ *by the* $\geq$*-rule, or (*ii*)* $\lambda(t.i)$ *has been obtained by applying the* $\geq$*-rule to* $\geq n\, R.C(s) \in \lambda(t)$*, and* $(\gamma_{i,1}, \ldots, \gamma_{i,n})$ *is the* i*-th* n*-tuple of* $\mathsf{iexp}(\geq n\, R.C, s, \mathcal{A})$ *such that, for each* $1 \leq j \leq n$*, either* $\gamma_{i,j} = *$ *or* $\gamma_{i,j}$ *is a named individual not occurring in* $\lambda(t)$*. The derivation* D *is* safe *if every nonleaf safe node has at least one safe child.*

Definition 3 might seem unnecessarily complex: soundness is ensured if we require each $\mathsf{iexp}(\geq n\, R.C, s, \mathcal{A})$ to contain a tuple $(*, \ldots, *)$, thus always allowing for the standard existential expansion. This is, however, unnecessarily restrictive. Consider the derivation shown in Figure 3, in which the $\geq$-rule is applied to $\mathcal{A}_\epsilon$, deriving $\mathcal{A}_1$ via individual reuse and $\mathcal{A}_2$ using the standard expansion. The first derivation is possibly unsound, so the second derivation is necessary in order to guarantee soundness; but then, there is no need to enforce soundness in the inferences below $\mathcal{A}_1$: applications of the $\geq$-rule below $\mathcal{A}_1$ can be performed in a way that can, but does not need to include $(*, \ldots, *)$ in $\mathsf{iexp}(\geq n\, R.C, s, \mathcal{A})$. Definition 3 formalizes this intuition: starting from the root ϵ, each derivation node that is obtained via a sound sequence of inferences must have at least one child obtained by a sound inference. The strategy that we present in Section 5.2 uses this definition: the first time a "reuse target" t is introduced, this is done nondeterministically; however, all subsequent reuses of t are deterministic. In other words, once our strategy nondeterministically decides to reuse t, it stays committed to this choice. We now present the main result of this section.

Theorem 1. *For* $\mathcal{C}$ *a set of HT-clauses,* $\mathcal{A}$ *an input ABox, and* iexp *a bounded expansion strategy, (1) each derivation for* $\mathcal{C}$*,* $\mathcal{A}$*, and* iexp *is finite; (2) if a successful derivation for* $\mathcal{C}$*,* $\mathcal{A}$*, and* iexp *exists, then* $(\mathcal{C}, \mathcal{A})$ *is satisfiable; and (3) if* $(\mathcal{C}, \mathcal{A})$ *is satisfiable, then each safe derivation for* $\mathcal{C}$*,* $\mathcal{A}$*, and* iexp *is successful.*

As discussed in [11], the *NI*-rule is not needed on HT-clauses obtained from $\mathcal{SHIQ}$ knowledge bases. With individual reuse, however, this is not the case: reusing a root individual c when expanding $\exists R.C(s)$ in Figure 2 can clearly trigger the *NI*-rule if c must satisfy an at-most restriction on R^-.

The *NI*-rule can introduce a performance penalty in practice, so we identify the class of *NI*-free strategies on which the rule is not needed even with individual reuse. Intuitively, an *NI*-free strategy can satisfy existential restrictions on root individuals either by introducing fresh root individuals or by reusing existing ones; however, it always applies the standard expansion for blockable individuals. For example, such a strategy might at first introduce and reuse only root individuals; if this proves ineffective and the models get large, it might then decide not to reuse individuals in further applications of the $\geq$-rule.

Definition 4 (*NI*-free Strategies). *An expansion strategy* iexp *is NI*-free *if* $\mathsf{iexp}(\geq n\, R.C, s, \mathcal{A}) = [(*, \ldots, *)]$ *whenever* s *is a blockable individual.*

Proposition 1. *In a derivation where* $\mathcal{C}$ *is a set of HT-clauses not containing equalities of the form* $x \approx y_i$ *and* $y_i \approx x$, $\mathcal{A}$ *is an ABox, and* iexp *is an NI-free expansion strategy, the precondition of the NI-rule is never satisfied.*

5 Two Expansion Strategies

5.1 The $\mathcal{ELOH}$ Case

In this section we present an expansion strategy for the DL $\mathcal{ELOH}$—a logic obtained from $\mathcal{EL}$++ [1] by disallowing role composition axioms $R \circ S \sqsubseteq T$ and concrete domains. We leave these constructors out because they are not available in $\mathcal{SHOIQ}$; however, our results can be straightforwardly extended to full $\mathcal{EL}$++. The expansion strategy for $\mathcal{ELOH}$ will provide us with an intuition on how to approach the general case. Furthermore, these results establish a relationship between model construction algorithms and the implication sets reasoning algorithm from [1]. Thus, tableau DL reasoners can be easily modified to obtain a worst-case optimal decision procedure for $\mathcal{ELOH}$ knowledge bases simply by choosing a suitable individual reuse strategy.

We start by defining $\mathcal{ELOH}$-clauses—the fragment of HT-clauses on which the $\mathcal{ELOH}$-strategy guarantees soundness. By inspecting the translation operator Ξ from [11, Section 4.1], it is easy to see that, if $\mathcal{K}$ is an $\mathcal{ELOH}$ knowledge base, then $\Xi_{\mathcal{TR}}(\mathcal{K})$ is a set of $\mathcal{ELOH}$-clauses.

Definition 5 ($\mathcal{ELOH}$-Clause). *An* $\mathcal{ELOH}$*-clause is an HT-clause* r *of the form* (2), (3), (4), *or* (5) *such that* $y_i \neq y_j$ *for* $i \neq j$; R_j *and* S *are atomic roles;* C *is of the form* $\bot$, A, *or* $\exists S.A$; A, A_i, *and* B_j *are either* $\top$ *or atomic but not nominal guard concepts; and* O_a *is a nominal guard concept.*

$$\bigwedge A_i(x) \wedge \bigwedge [R_j(x, y_j) \wedge B_j(y_j)] \rightarrow C(x) \tag{2}$$

$$O_a(y_a) \wedge \bigwedge A_i(x) \wedge \bigwedge [R_j(x, y_j) \wedge B_j(y_j)] \rightarrow x \approx y_a \tag{3}$$

$$\bigwedge A_i(x) \wedge \bigwedge [R_j(x, y_j) \wedge B_j(y_j)] \rightarrow S(x, y_j) \tag{4}$$

$$O_a(y_a) \wedge \bigwedge A_i(x) \wedge \bigwedge [R_j(x, y_j) \wedge B_j(y_j)] \rightarrow S(x, y_a) \tag{5}$$

We now define the expansion strategy for $\mathcal{ELOH}$.

Definition 6 ($\mathcal{ELOH}$-Strategy). *We assume that, for A an atomic concept, $\top$, or $\bot$, the set of named individuals N_I contains a distinct* representative in*dividual α_A. The $\mathcal{ELOH}$-strategy is defined as* $\mathsf{iexp}_{EL}(\exists R.A, s, \mathcal{A}) = [(\|\alpha_A\|_{\mathcal{A}})]$, *for each concept $\exists R.A$, individual s, and ABox $\mathcal{A}$.*

Since iexp_{EL} deterministically reuses $\|\alpha_A\|_{\mathcal{A}}$, derivations created using this strategy are typically not safe, so Theorem 1 does not apply. Nonetheless, the strategy always produces sound derivations on $\mathcal{ELOH}$-clauses.

Theorem 2. *For $\mathcal{C}$ a set of $\mathcal{ELOH}$-clauses and $\mathcal{A}$ an initial ABox not containing representative individuals, if $(\mathcal{C}, \mathcal{A})$ is satisfiable, then each derivation for $\mathcal{C}$, $\mathcal{A}$, and* iexp_{EL} *is successful.*

This theorem can be intuitively understood as follows. Let $\mathcal{C}$ be a set of DL-clauses of the form (2) (we do not consider other types for simplicity) and $\mathcal{A}$ an ABox such that $(\mathcal{C}, \mathcal{A})$ is satisfiable. Assume now that we apply the standard hypertableau calculus without individual reuse and *without blocking* to $\mathcal{C}$ and $\mathcal{A}$. Since blocking is not used, and because all DL-clauses from $\mathcal{C}$ are Horn clauses, this will produce a possibly infinite *canonical* ABox $\mathcal{A}_\infty$ which naturally defines a model of $(\mathcal{C}, \mathcal{A})$. Consider now how individuals are introduced into $\mathcal{A}_\infty$. In particular, assume that an assertion $\exists R.A(s)$ is expanded into assertions $R(s,t)$ and $A(t)$ for t a fresh successor of s. The key observation is that, if the *Hyp*-rule were to additionally derive $C(t)$, then this assertion depends only on the successors of t: a DL-clause of the form (2) can only check properties of successors of the individual mapped to x and not of its predecessors. In other words, the $\mathcal{ELOH}$-clauses do not allow for DL-clauses such as $R(y,x) \rightarrow C(x)$, which might derive C for an individual x based on the properties of its predecessor y. Thus, if in some other part of the model construction we expand $\exists S.A(u)$ into $S(u,v)$ and $A(v)$, the assertions derived for v will depend only on the successors of v. The application of the derivation rules to t and v and their descendants is thus fully determined by the initial "seed" concept A, so t and v will occur in $\mathcal{A}_\infty$ in exactly the same concept assertions (i.e., $C(t) \in \mathcal{A}_\infty$ iff $C(v) \in \mathcal{A}_\infty$). There is, therefore, no need to create distinct individuals t and v at all: we can fold $\mathcal{A}_\infty$ into a model that contains exactly one individual α_A for each "seed" concept A. Effectively, if $(\mathcal{C}, \mathcal{A})$ is satisfiable, then it has such a folded model. It is easy to see that iexp_{EL} essentially constructs exactly this folded model.

The number of representative individuals is linear in the number of the atomic concepts, so the ABoxes constructed in a derivation are polynomial in size. To obtain a polynomial decision procedure, we must be careful in how we apply the *Hyp*-rule to $\mathcal{ELOH}$-clauses: with i individuals and v variables in a DL-clause, a naïve application of the *Hyp*-rule (for each x, for each y_1, for each y_2, and so on) examines i^v combinations. Note, however, that y_i occur in the antecedent only in a tree-like way and that they do not occur in the consequent; hence, we can match each y_i independently, thus giving rise to $v \cdot i$ combinations.

Theorem 3. *A derivation for a set of $\mathcal{ELOH}$-clauses $\mathcal{C}$, an initial ABox $\mathcal{A}$ not containing representative individuals, and* iexp_{EL} *can be constructed in time polynomial in* $|\mathcal{C}, \mathcal{A}|$.

5.2 The General Case

In this section we define an expansion strategy that is applicable to $\mathcal{SHOIQ}$. The strategy tries to mimic the $\mathcal{ELOH}$ case: it expands $\exists R.A(s)$ into $R(s, \alpha_A)$ and $A(\alpha_A)$ for α_A the representative individual, but it also allows for the default expansion $(*)$ as well. Our idea is that, whenever u and v are introduced by expanding concepts $\exists R.A$ and $\exists S.A$, respectively, the individuals u and v should not differ greatly—that is, the derivation rules applied to u and v should be largely the same. For example, it is reasonable to expect that all individuals representing a heart in a model of GALEN should have the same properties. This basic idea has been refined in several ways.

Our experiments have shown that assertions of the form $\exists R.A(s)$, where A is a concept introduced by the structural transformation in the preprocessing phase, are best expanded without individual reuse. Such concepts correspond to complex formulae identifying sets of objects with certain properties, so they are unlikely to have singleton extensions.

Our experiments have also shown that, given $C \sqsubseteq \exists R.D$ and $D \sqsubseteq \exists S.C$, the instance of C implied by the second axiom is usually the instance of C from the first axiom. For example, in GALEN "each artery has a tunica intima as its part," and "each tunica intima is a layer of an artery"; clearly, the second artery is the same as the first one. Therefore, when expanding $\exists S.C(s)$, our strategy tries to reuse the parent t of s if $C(t)$ has already been derived.

DL knowledge bases often contain concepts that should be expanded without reuse—that is, whose expansion using representative individuals usually fails. This can introduce unnecessary nondeterminism, so our strategy learns from failed choices: if an expansion of $\exists R.A$ by reusing α_A fails, our strategy does not reuse α_A in future expansions. The main problem is, once a clash is detected, to determine whether individual reuse caused the inconsistency and, if so, which individual α_A is the culprit. To solve this problem, we construct derivations by depth-first search with dependency-directed backtracking [2, Chapter 9]—a search technique used in most tableau-based reasoners. Roughly speaking, each nondeterministic choice in the derivation is numbered, and each assertion is annotated with a set of choices it depends on. When $\bot$ is derived, the set of choices $S_\bot$ associated with $\bot$ tells us which choices are conflicting. Let $m_\bot$ be the maximal choice from $S_\bot$. Backtracking any choice below $m_\bot$ will invariably lead to a clash; thus, $m_\bot$ identifies the failed choice at which we can safely continue the search. If $m_\bot$ corresponds to a nondeterministic expansion of $\exists R.A$ by reusing α_A, then we consider the reuse of α_A as failed; therefore, we add A to a special *no-good* set of concepts, which ensures that α_A is not reused in future expansions of concepts of the form $\exists S.A$.

Definition 7. *We assume that derivations are constructed by depth-first search and dependency-directed backtracking [2, Chapter 9], and that a set NG of* no-good concepts *is maintained in the process as discussed next. For R a role, B a possibly negated atomic concept, s an individual, and $\mathcal{A}$ an ABox, the* general strategy iexp_{DL} *returns the first result from the following list for which the condition is satisfied.*

- $\mathsf{iexp}_{DL}(\exists R.B, s, \mathcal{A}) = [(t), (*)]$ *if t is the parent of s and $B(t) \in \mathcal{A}$.*
- $\mathsf{iexp}_{DL}(\exists R.B, s, \mathcal{A}) = [(*)]$ *if B has been introduced by the structural transformation during preprocessing.*
- $\mathsf{iexp}_{DL}(\exists R.B, s, \mathcal{A}) = [(\alpha_B), (*)]$ *if α_B does not occur in $\mathcal{A}$ and B is not contained in the current set NG. Whenever an expansion using α_B is backtracked as per dependency-directed backtracking, the concept B is added to the no-good set NG.*
- $\mathsf{iexp}_{DL}(\exists R.B, s, \mathcal{A}) = [(\alpha_B)]$ *if α_B occurs in $\mathcal{A}$.*
- $\mathsf{iexp}_{DL}(\geq n\, R.B, s, \mathcal{A}) = [(*, \ldots, *)]$ *in all other cases.*

Clearly, iexp_{DL} is bounded. Furthermore, the first time $\exists R.B$ is expanded, iexp_{DL} nondeterministically introduces α_B as a target for future reuse; in all subsequent expansions of $\exists S.B$, iexp_{DL} stays committed to this choice and reuses α_B deterministically. These observations allow us to prove the following proposition.

Proposition 2. *For $\mathcal{C}$ a set of HT-clauses and $\mathcal{A}$ an initial ABox not containing representative individuals, each derivation for $\mathcal{C}$, $\mathcal{A}$, and iexp_{DL} is safe.*

6 Evaluation

We have implemented individual reuse in HermiT, and have compared its performance with FaCT++ v1.1.10 [16] and Pellet 1.5.1 [12]. When parameterized with iexp_{EL}, the hypertableau calculus becomes quite similar to the implication sets algorithm [1], so any difference in the performance between HermiT and the $\mathcal{EL}$++-specific reasoner CEL is likely to be caused by engineering, rather than algorithmic issues. Therefore, we focus here on evaluating iexp_{DL}.

We have selected several test ontologies commonly used in practice, and have tried to classify them (i.e., to compute $\mathcal{K} \models C \sqsubseteq D$ for all atomic concepts C and D) using HermiT with and without individual reuse, FaCT++, and Pellet. The results are summarized in Table 3. Most unsuccessful tests failed because the reasoners ran out of memory. Several tests were interrupted after 30 minutes; in all such cases, the reasoner got stuck in the first nontrivial subsumption test, so it is unlikely that more time would allow the reasoner to complete the classification. Tests were conducted on a standard laptop PC with 1 GB of RAM running Windows XP. We used Java 1.6.0_04 for Java-based reasoners allowing 600 MB of heap space in each test. All ontologies are available from HermiT's Web page.

Wine. The Wine ontology has often been used to demonstrate the features of description logics. Classifying it was initially a challenge for tableau reasoners, but current reasoners can routinely process it.

Table 3. Summary of Test Results

Ontology	HermiT (with reuse)	HermiT (no reuse)	FaCT++	Pellet
Wine	classified in 37 s	classified in 36 s	classified in 316 s	classified in 25 s
FMA-fragment	classified in 680 s	—	—	—
BAMS	classified in 19 s	—	—	—
DOLCE	classified in 103 s	—	classified in 12 s	—
GALEN-original	classified in 275 s	classified in 26 s	—	—
GALEN-module	—	—	—	—
GALEN-module-no-inv	solves individual tests	—	—	—

FMA-fragment and BAMS. FMA [14] is a large ontology that was not originally developed using a DL, and various translations of different fragments of FMA into DLs have been produced. We used a translation from the Bio-Health Informatics Group at the University of Manchester; it contains 6,487 atomic concepts, 165 atomic roles, 98 individuals, and 18,678 axioms in total. The Brain Architecture Management System (BAMS) is a brain anatomy ontology produced at the University of Southern California. We used the translation of the 1998 version of BAMS into OWL, which contains 1,110 atomic concepts, 13 atomic roles, 999 individuals, and 20,176 axioms in total. Both ontologies are nontrivial, as they use inverse and functional roles, disjunctions, and nominals.

Without individual reuse, no tool can classify either ontology: reasoners end up constructing large models and quickly exhaust the available memory. In contrast, individual reuse makes these ontologies easy: the models constructed by HermiT consist of a couple of thousands of individuals at most.

DOLCE. DOLCE[3] is a foundational ontology developed in the WonderWeb project, and it is quite complex: it uses disjunctions, nominals, and number restrictions. It is, however, relatively small, containing 211 atomic concepts, 317 atomic roles, 39 individuals, and 1,797 axioms in total. As Table 3 shows, FaCT++ can process DOLCE even without individual reuse, whereas Pellet and HermiT cannot. We conjecture that this is due to ordering optimizations [15], which can have a significant impact on the performance of reasoning. HermiT can process DOLCE provided that individual reuse is turned on. This suggests that individual reuse might compensate for the lack of an optimal ordering, which can be difficult to find in practice.

GALEN. GALEN-original is a version of GALEN dating from about 10 years ago. HermiT is currently the only reasoner that can classify this ontology, and it can do so even without individual reuse [10]. The good performance of HermiT in this case is mainly due to the fact that, after computing a model, HermiT caches the labels of all unblocked individuals and uses them subsequently as potential blockers [11]. Thus, only the first test is hard (it takes about 90% of the

[3] `http://www.loa-cnr.it/DOLCE.html`

total time), and all subsequent tests are easy due to caching. With individual reuse, however, this optimization is ineffective, as most individuals in the model are root individuals that cannot be used as blockers. Each subsumption test takes typically an order of magnitude less time with individual reuse than the "hard" test without reuse; however, the cumulative slowdown due to the lack of caching is detrimental. Caching should be possible, however, with individual reuse as well: we can cache the model constructed in different runs and reuse individuals from it as needed. We shall investigate such a technique in our future work.

GALEN was significantly extended in the past 10 years, and the current version of it consists of more than 38,000 concepts, so we have extracted a module from it using the algorithm from [7]. The module is still quite large: it contains 6,362 atomic concepts, 162 atomic roles, and 14,783 axioms in total. None of the reasoners were able to deal with the GALEN-module. We therefore tried removing all axioms of the form $R \sqsubseteq S^-$, effectively eliminating the link between a role and its inverse. After this transformation, HermiT was able to solve individual subsumption tests; however, each test took about 40 s, which made classification of the entire ontology infeasible. We believe that this can be significantly improved using the above mentioned caching technique.

Individual reuse is ineffective on GALEN mainly because the ontology contains numerous functional and inverse-functional roles, which makes identifying concepts suitable for individual reuse difficult. For example, the axiom "each nerve has exactly one axon and vice versa" clearly prevents the reuse of the axon concept. We tried to address this problem by selecting manually the concepts that should be reused, but even this was to no avail. In fact, it seems to us that inverse properties in GALEN are sometimes overconstrained (e.g., w.r.t. functionality). The tools used to develop GALEN largely ignore the semantics of inverse roles, so modeling errors of this kind might easily have gone unnoticed.

7 Conclusion

We have presented a novel calculus for DL reasoning based on individual reuse: instead of always introducing a fresh individual in order to satisfy an existential restriction, our calculus tries to reuse an individual from a model created thus far. Furthermore, we have shown that individual reuse can be done deterministically for $\mathcal{ELOH}$—an expressive but yet tractable DL. Finally, we have implemented our calculus in the reasoner HermiT and have evaluated its performances. The empirical results suggest that individual reuse can mean the difference between success and failure on practical ontologies. The main challenge for our future research is to devise an expansion strategy that will allow us to classify GALEN—the nemesis of DL reasoners.

References

1. Baader, F., Brandt, S., Lutz, C.: Pushing the $\mathcal{EL}$ Envelope. In: Pack Kaelbling, L., Saffiotti, A. (eds.) Proc. of the 19th Int. Joint Conference on Artificial Intelligence (IJCAI 2005), Edinburgh, UK, July 30–August 5 2005, pp. 364–369. Morgan Kaufmann, San Francisco (2005)
2. Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P.F. (eds.): The Description Logic Handbook: Theory, Implementation and Applications, 2nd edn. Cambridge University Press, Cambridge (2007)
3. Baader, F., Lutz, C., Suntisrivaraporn, B.: CEL—A Polynomial-Time Reasoner for Life Science Ontologies. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 287–291. Springer, Heidelberg (2006)
4. Baumgartner, P., Furbach, U., Niemelä, I.: Hyper Tableaux. In: Orłowska, E., Alferes, J.J., Moniz Pereira, L. (eds.) JELIA 1996. LNCS, vol. 1126, pp. 1–17. Springer, Heidelberg (1996)
5. Bry, F., Torge, S.: A Deduction Method Complete for Refutation and Finite Satisfiability. In: Dix, J., del Cerro, L.F., Furbach, U. (eds.) JELIA 1998. LNCS (LNAI), vol. 1489, pp. 122–138. Springer, Heidelberg (1998)
6. Georgieva, L., Hustadt, U., Schmidt, R.A.: Hyperresolution for Guarded Formulae. Journal of Symbolic Computation 36(1-2), 163–192 (2003)
7. Cuenca Grau, B., Horrocks, I., Kazakov, Y., Sattler, U.: Just the Right Amount: Extracting Modules from Ontologies. In: Proc. of the 16th Int. Conf. on World Wide Web (WWW 2007), Banff, AB, Canada, May 8–12, 2007, pp. 717–726. ACM Press, New York (2007)
8. Haarslev, V., Möller, R.: RACER System Description. In: Goré, R.P., Leitsch, A., Nipkow, T. (eds.) IJCAR 2001. LNCS (LNAI), vol. 2083, pp. 701–706. Springer, Heidelberg (2001)
9. Hustadt, U., Schmidt, R.A.: Issues of Decidability for Description Logics in the Framework of Resolution. In: Caferra, R., Salzer, G. (eds.) FTP 1998. LNCS (LNAI), vol. 1761, pp. 191–205. Springer, Heidelberg (2000)
10. Motik, B., Shearer, R., Horrocks, I.: Optimized Reasoning in Description Logics using Hypertableaux. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 67–83. Springer, Heidelberg (2007)
11. Motik, B., Shearer, R., Horrocks, I.: Hypertableau Reasoning for Description Logics. Technical report, University of Oxford, Submitted to an international journal (2008)
12. Parsia, B., Sirin, E.: Pellet: An OWL-DL Reasoner. In: McIlraith, S.A., Plexousakis, D., van Harmelen, F. (eds.) ISWC 2004. LNCS, vol. 3298. Springer, Heidelberg (2004)
13. Rector, A.L., Nowlan, W.A., Glowinski, A.: Goals for concept representation in the galen project. In: Safran, C. (ed.) Proc. of the 17th Annual Symposium on Computer Applications in Medical Care (SCAMC 1993), Washington DC, USA, November 1–3 1993, pp. 414–418. McGraw-Hill, New York (1993)
14. Rosse, C., Mejino, J.V.L.: A reference ontology for biomedical informatics: the Foundational Model of Anatomy. Journal of Biomedical Informatics 36, 478–500 (2003)
15. Tsarkov, D., Horrocks, I.: Ordering Heuristics for Description Logic Reasoning. In: Pack Kaelbling, L., Saffiotti, A. (eds.) Proc. of the 19th Int. Joint Conf. on Artificial Intelligence (IJCAI 2005), Edinburgh, UK, July 30–August 5 2005, pp. 609–614. Morgan Kaufmann Publishers, San Francisco (2005)

16. Tsarkov, D., Horrocks, I.: FaCT++ Description Logic Reasoner: System Description. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 292–297. Springer, Heidelberg (2006)
17. Vardi, M.Y.: Why Is Modal Logic So Robustly Decidable. In: Immerman, N., Kolaitis, P. (eds.) Proc. of a DIMACS Workshop on Descriptive Complexity and Finite Models, January 14–17, 1996. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 31, pp. 149–184. Princeton University Press, American Mathematical Society (1996)

The Logical Difference Problem for Description Logic Terminologies

Boris Konev, Dirk Walther, and Frank Wolter

University of Liverpool, Liverpool, UK
{konev,dwalther,wolter}@liverpool.ac.uk

Abstract. We consider the problem of computing the logical difference between distinct versions of description logic terminologies. For the lightweight description logic $\mathcal{EL}$, we present a tractable algorithm which, given two terminologies and a signature, outputs a set of concepts, which can be regarded as the logical difference between the two terminologies. As a consequence, the algorithm can also decide whether they imply the same concept implications in the signature. A prototype implementation CEX of this algorithm is presented and experimental results based on distinct versions of SNOMED CT, the Systematized Nomenclature of Medicine, Clinical Terms, are discussed. Finally, results regarding the relation to uniform interpolants and possible extensions to more expressive description logics are presented.

1 Introduction

The standard diff operation for text files is an indispensable tool for comparing different versions of texts, and similar operations are available to software engineers comparing distinct versions of code produced in collaborative software projects. As observed, e.g., in [14], such a purely syntactic diff operation is hardly useful if the text consists of a set of axioms of an ontology. In this case, one is usually not interested in a comparison of the syntactic form of axioms, but in the consequences that the ontologies have. The authors of [14] present a number of heuristic rules to address this problem and develop a diff operator for ontologies. Except theoretical results in [12,13,9], we are not aware of any logic-based approach to computing the logical diff of ontologies.

Our formalisation of the logical difference problem is based on the observation that when comparing distinct versions of ontologies one should take into account their signatures. In fact, the interesting differences between ontologies are those formulated in their shared signature (or even subsets thereof), and not those involving symbols used only in one of the two ontologies. Thus, the proposed notion of logical difference is based on the notion of *Σ-entailment*: an ontology $\mathcal{T}$ Σ-entails an ontology $\mathcal{T}'$ for a signature Σ, if for all concept implications $C \sqsubseteq D$ in Σ, $\mathcal{T}' \models C \sqsubseteq D$ implies $\mathcal{T} \models C \sqsubseteq D$. If $\mathcal{T}$ and $\mathcal{T}'$ mutually Σ-entail each other, then they are called *Σ-inseparable*. By taking Σ as the set of shared symbols of $\mathcal{T}$ and $\mathcal{T}'$, Σ-inseparability means that $\mathcal{T}$ and $\mathcal{T}'$ are not distinguishable by means of concept implications in their shared signature. In this case, their logical difference will be regarded as empty.

We show that deciding Σ-entailment is tractable for $\mathcal{EL}$-terminologies, i.e., sets of possibly cyclic concept definitions in the lightweight description logic $\mathcal{EL}$; see [1,10]. Observe that for ontologies formulated as general TBoxes in description logics, the computational complexity of deciding Σ-entailment is by at least one exponential harder than the deduction problem, e.g., it is 2ExpTime-complete for expressive description logics such as $\mathcal{ALC}$, $\mathcal{ALCQ}$, and $\mathcal{ALCQI}$ [6,12] and ExpTime-complete for $\mathcal{EL}$ itself [13]. Moreover, even in such simple formalisms as acyclic propositional Horn Logic Σ-entailment is co-NP-complete [5].

In applications, it is not enough to decide whether two ontologies are logically different, but an informative list of differences is required. We show that for any concept implication $C \sqsubseteq D$ in the logical difference between two $\mathcal{EL}$-terminologies, there exist subconcepts C' and D' of C and D, respectively, such that $C' \sqsubseteq D'$ is in the logical difference and C' or D' is a concept name. Thus, listing the set of all concept names involved in such implications appears to be an informative approximation of the logical difference between two $\mathcal{EL}$-terminologies. This list is empty if, and only if, there is no logical difference between the two terminologies.

The system CEX implements, by employing a dynamic programming approach, the algorithm deciding Σ-entailment and lists the set of logical differences described above for acyclic $\mathcal{EL}$-terminologies. We present a variety of experiments in which CEX is applied to different versions of Snomed ct, the Systematized Nomenclature of Medicine, Clinical Terms. This terminology comprises $\sim$0.4 million terms and underlies the systematised medical terminology used in the health systems of the US, the UK, and other countries [17].

Finally, we discuss an alternative approach to deciding Σ-entailment using uniform interpolants and explore the complexity of corresponding reasoning problems for acyclic $\mathcal{ALC}$-terminologies.

Detailed proofs are provided in the technical report [11].

2 Preliminaries

Let $\mathsf{N_C}$ and $\mathsf{N_R}$ be countably infinite and disjoint sets of *concept names* and *role names*, respectively. In the description logic $\mathcal{EL}$, *concepts* C are built according to the syntax rule

$$C ::= \top \mid A \mid C \sqcap D \mid \exists r.C,$$

where A ranges over $\mathsf{N_C}$, r ranges over $\mathsf{N_R}$, and C, D range over concepts. The semantics of concepts is defined by means of *interpretations* $\mathcal{I} = (\Delta^\mathcal{I}, \cdot^\mathcal{I})$, where the interpretation *domain* $\Delta^\mathcal{I}$ is a non-empty set, and $\cdot^\mathcal{I}$ is a function mapping each concept name A to a subset $A^\mathcal{I}$ of $\Delta^\mathcal{I}$ and each role name $r^\mathcal{I}$ to a binary relation $r^\mathcal{I} \subseteq \Delta^\mathcal{I} \times \Delta^\mathcal{I}$. The function $\cdot^\mathcal{I}$ is inductively extended to arbitrary concepts by setting $\top^\mathcal{I} := \Delta^\mathcal{I}$, $(C \sqcap D)^\mathcal{I} := C^\mathcal{I} \cap D^\mathcal{I}$, and $(\exists r.C)^\mathcal{I} := \{d \in \Delta^\mathcal{I} \mid \exists e \in C^\mathcal{I} : (d, e) \in r^\mathcal{I}\}$.

A *general TBox* is a finite set of *axioms*, where an axiom can be either a *concept inclusion (CI)* $C \sqsubseteq D$ or a *concept equality (CE)* $C \equiv D$, where C, D

are concepts. An interpretation $\mathcal{I}$ *satisfies* a CI $C \sqsubseteq D$ (written $\mathcal{I} \models C \sqsubseteq D$) if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$; it *satisfies* a CE $C \equiv D$ (written $\mathcal{I} \models C \equiv D$) if $C^{\mathcal{I}} = D^{\mathcal{I}}$. $\mathcal{I}$ is a *model* of a general TBox $\mathcal{T}$ if it satisfies all axioms in $\mathcal{T}$. We write $\mathcal{T} \models C \sqsubseteq D$ ($\mathcal{T} \models C \equiv D$) if every model of $\mathcal{T}$ satisfies $C \sqsubseteq D$ ($C \equiv D$, respectively).

Our main concern in this paper are *terminologies*, i.e., general TBoxes $\mathcal{T}$ satisfying the following two conditions:

– $\mathcal{T}$ consists of CEs of the form $A \equiv C$ (*concept definitions*) and CIs of the form $A \sqsubseteq C$ (*primitive concept definitions*) only, where A is a concept name;
– no concept name occurs more than once on the left hand side of an axiom in $\mathcal{T}$.

Define the relation $\prec_{\mathcal{T}}$ between concept names by setting $A \prec_{\mathcal{T}} B$ if there exists an axiom of the form $A \equiv C$ or $A \sqsubseteq C$ in $\mathcal{T}$ such that B occurs in C. A terminology $\mathcal{T}$ is called *acyclic* if the transitive closure $\prec_{\mathcal{T}}^{*}$ of $\prec_{\mathcal{T}}$ is irreflexive.

A *signature* Σ is a finite subset of $\mathsf{N_C} \cup \mathsf{N_R}$. The signature $\mathsf{sig}(C)$ ($\mathsf{sig}(\alpha)$, $\mathsf{sig}(\mathcal{T})$) of a concept C (axiom α, terminology $\mathcal{T}$) is the set of concept and role names which occur in C (α, $\mathcal{T}$, respectively). If $\mathsf{sig}(C) \subseteq \Sigma$, we also call C a *Σ-concept* and similarly for axioms and terminologies.

Definition 1 (Σ-difference, Σ-entailment). *Let $\mathcal{T}$ and $\mathcal{T}'$ be terminologies and Σ a signature. The Σ-difference,* $\mathsf{Diff}_{\Sigma}(\mathcal{T}, \mathcal{T}')$, *between $\mathcal{T}$ and $\mathcal{T}'$ is defined as*

$$\mathsf{Diff}_{\Sigma}(\mathcal{T}, \mathcal{T}') = \{C \sqsubseteq D \mid \mathcal{T} \not\models C \sqsubseteq D \textit{ and } \mathcal{T}' \models C \sqsubseteq D \textit{ and } \mathsf{sig}(C \sqsubseteq D) \subseteq \Sigma\}.$$

$\mathcal{T}$ Σ-entails $\mathcal{T}'$ if, and only if, $\mathsf{Diff}_{\Sigma}(\mathcal{T}, \mathcal{T}') = \emptyset$. *$\mathcal{T}$ and $\mathcal{T}'$ are called Σ-inseparable if $\mathcal{T}$ and $\mathcal{T}'$ Σ-entail each other.*

Example 1. Observe that, in some cases, $\mathsf{Diff}_{\Sigma}(\mathcal{T}, \mathcal{T}')$ only contains concept implications of at least exponential size, even for acyclic terminologies. To start with, let $\mathcal{T} = \emptyset$,

$$\mathcal{T}' = \{A_0 \sqsubseteq B_0, A_1 \equiv B_n\} \cup \{B_{i+1} \equiv \exists r.B_i \sqcap \exists s.B_i \mid 0 \leq i < n\},$$

and $\Sigma = \{A_0, A_1, r, s\}$. Then $\mathcal{T}'$ is not Σ-entailed by $\mathcal{T}$, and a minimal implication of the form $C \sqsubseteq A_1$ in $\mathsf{Diff}_{\Sigma}(\mathcal{T}, \mathcal{T}')$ is given by $C_n \sqsubseteq A_1$, where $C_0 = A_0$ and $C_{i+1} = \exists r.C_i \sqcap \exists s.C_i$, for $i \geq 0$. Clearly, C_n is of exponential size. Observe, however, that there exist much smaller implications than $C_n \sqsubseteq A_1$ in $\mathsf{Diff}_{\Sigma}(\mathcal{T}, \mathcal{T}')$. Namely, $A_1 \sqsubseteq \exists r.\top$, $A_1 \sqsubseteq \exists s.\top$, $A_1 \sqsubseteq \exists r.\top \sqcap \exists s.\top$, etc. To avoid this type of implications in $\mathsf{Diff}_{\Sigma}(\mathcal{T}, \mathcal{T}')$ replace $\mathcal{T}$ by

$$\mathcal{T}_0 = \{A_1 \sqsubseteq F_0\} \cup \{F_i \sqsubseteq \exists r.F_{i+1} \sqcap \exists s.F_{i+1} \mid 0 \leq i < n\}.$$

Then one can easily see that $C_n \sqsubseteq A_1$ is the smallest implication in $\mathsf{Diff}_{\Sigma}(\mathcal{T}_0, \mathcal{T}')$. Observe, however, that if we use structure sharing and define the size of C_n as the number of its subconcepts, then C_n is only of polynomial size.

Observe that if $\mathcal{T}$ Σ-entails $\mathcal{T}'$, then $\mathcal{T}$ Σ'-entails $\mathcal{T}'$ for any Σ' with $\Sigma' \cap \mathsf{sig}(\mathcal{T}') \subseteq \Sigma$. This follows immediately from the following interpolation result [16].

$$\frac{}{C \sqsubseteq C}\ (\textsc{Ax}) \quad \frac{}{C \sqsubseteq \top}\ (\textsc{AxTop}) \quad \frac{C \sqsubseteq E}{C \sqcap D \sqsubseteq E}\ (\textsc{AndL1}) \quad \frac{D \sqsubseteq E}{C \sqcap D \sqsubseteq E}\ (\textsc{AndL2})$$

$$\frac{C \sqsubseteq E \quad C \sqsubseteq D}{C \sqsubseteq D \sqcap E}\ (\textsc{AndR}) \quad \frac{C \sqsubseteq D}{\exists r.C \sqsubseteq \exists r.D}\ (\textsc{Ex})$$

$$\frac{C_A \sqsubseteq D}{A \sqsubseteq D}\ (\textsc{DefL}) \quad \frac{D \sqsubseteq C_A}{D \sqsubseteq A}\ (\textsc{DefR}) \quad \text{where } A \equiv C_A \in \mathcal{T}$$

$$\frac{C_A \sqsubseteq D}{A \sqsubseteq D}\ (\textsc{PDefL}) \quad \text{where } A \sqsubseteq C_A \in \mathcal{T}$$

Fig. 1. Gentzen-style proof system for $\mathcal{EL}$ terminologies

Theorem 1. *$\mathcal{EL}$ has the interpolation property, i.e., if $\mathcal{T} \models C \sqsubseteq D$, then there exists a finite set $\mathcal{T}_0$ of CIs with $\mathsf{sig}(\mathcal{T}_0) \subseteq \mathsf{sig}(\mathcal{T}) \cap \mathsf{sig}(C \sqsubseteq D)$ such that $\mathcal{T} \models \mathcal{T}_0$ and $\mathcal{T}_0 \models C \sqsubseteq D$.*

3 Basic Properties of $\mathcal{EL}$

We derive basic properties of $\mathcal{EL}$ from the Gentzen-style sequent calculus of Hofmann [10], see Figure 1.[1] The basic calculus of [10] considers $\mathcal{EL}$ without the constant $\top$ and for terminologies without primitive concept definitions. To take care of $\top$, we have added the rule (AxTop), and (PDefL) is the rule representing axioms of the form $A \sqsubseteq C$. Cut-elimination, completeness, and correctness can now be proved by a straightforward extension of the proof in [10]. For a terminology $\mathcal{T}$ and concepts C, D, we write $\mathcal{T} \vdash C \sqsubseteq D$ iff there exists a proof of $C \sqsubseteq D$ in the calculus of Figure 1.

Theorem 2 (Hofmann). *For all terminologies $\mathcal{T}$ and concepts C, D, it holds that $\mathcal{T} \models C \sqsubseteq D$ if, and only if, $\mathcal{T} \vdash C \sqsubseteq D$.*

We apply this calculus to derive a description of the syntactic form of concepts C such that $\mathcal{T} \models C \sqsubseteq D$, where D is not equivalent to a conjunction. Call a concept name A *primitive in $\mathcal{T}$* if A does not occur on the left hand side of an axiom in $\mathcal{T}$. Call A *pseudo-primitive in $\mathcal{T}$* if it is primitive in $\mathcal{T}$ or occurs on the left hand side of primitive concept definitions in $\mathcal{T}$. In what follows, we say that a concept F is a conjunction of concepts if $F = \bigsqcap_{D \in X} D$, for a set X of concepts. Any $D \in X$ is then called a *conjunct* of F and, if D is a concept name, then it is called an *atomic conjunct* of F. We sometimes write $D \in F$ instead of $D \in X$ and if X is empty, then F denotes the concept $\top$.

Lemma 1. *Let $\mathcal{T}$ be a terminology and $C = F \sqcap \bigsqcap_{(r,D) \in Q} \exists r.D$, where F is a conjunction of concept names and Q is a set of pairs (r, D) in which r is a role and D a concept.*

[1] Alternatively, one could start from the model-theoretic analysis of $\mathcal{EL}$ terminologies in [1].

1. *If $\mathcal{T} \models C \sqsubseteq A$ for an A which is pseudo-primitive in $\mathcal{T}$, then $\mathcal{T} \models B \sqsubseteq A$, for some atomic conjunct B of F.*
2. *If $\mathcal{T} \models C \sqsubseteq \exists s.C_0$, then*
   – *$\mathcal{T} \models B \sqsubseteq \exists s.C_0$, for some atomic conjunct B of F, or*
   – *there exists $(r, D) \in Q$ such that $r = s$ and $\mathcal{T} \models D \sqsubseteq C_0$.*

Proof. We use Theorem 2 and prove Point 1. Point 2 is proved similarly. Let $\mathcal{T} \vdash C \sqsubseteq A$, where A is pseudo-primitive in $\mathcal{T}$. Let $\mathcal{D}$ be a proof of $C \sqsubseteq A$. Note that, since A is pseudo-primitive in $\mathcal{T}$, $\mathcal{D}$ can only end with one of Ax, AndL1, AndL2, DefL, or PDefL. We show that then $\mathcal{T} \vdash B \sqsubseteq A$, for some conjunct B of F, by induction on the number n of conjuncts in C.

The base case of $n = 1$ is trivial: $\mathcal{D}$ can only end with one of Ax, PDefL, or DefL; so, C is a concept name itself.

Assume $n > 1$. Then $\mathcal{D}$ can only end with one of AndL1 or AndL2. In any case, there exists a conjunct C' of C such that $\mathcal{T} \vdash C' \sqsubseteq A$ and C' contains less conjuncts than C. By induction, there exists a concept name B which is a conjunct in C' such that $\mathcal{T} \vdash B \sqsubseteq A$. Note now that B is also a conjunct of C.

We apply Lemma 1 to show that if $\mathcal{T}$ does not Σ-entail $\mathcal{T}'$, then there exists $C \sqsubseteq D \in \mathsf{Diff}_\Sigma(\mathcal{T}, \mathcal{T}')$ such that C or D is a concept name.

Lemma 2. *Let $\mathcal{T}$ and $\mathcal{T}'$ be terminologies and Σ a signature. If $C \sqsubseteq D \in \mathsf{Diff}_\Sigma(\mathcal{T}, \mathcal{T}')$, then there exist subconcepts C' and D' of C and D, respectively, such that $C' \sqsubseteq D' \in \mathsf{Diff}_\Sigma(\mathcal{T}, \mathcal{T}')$ and $C' \sqsubseteq D'$ is of the form $A \sqsubseteq \exists r.D_0$ or $C_0 \sqsubseteq A$, where A is a concept name.*

Proof. Let $C \sqsubseteq D \in \mathsf{Diff}_\Sigma(\mathcal{T}, \mathcal{T}')$. Then $D \neq \top$ because otherwise $\mathcal{T} \models C \sqsubseteq D$. If $D = D_1 \sqcap D_2$, then one of $C \sqsubseteq D_i$, $i = 1, 2$, is in the Σ-difference. If $D = \exists r.D_1$ then, by Lemma 1, either there exists a subconcept A of C, A a concept name, such that $A \sqsubseteq D$ is in the Σ-difference, or there exists a subconcept $\exists r.C_1$ of C, such that $C_1 \sqsubseteq D_1$ is in the Σ-difference. Simplify $C \sqsubseteq D$ until none of these simplification rules is applicable. The resulting CI is as required.

4 Deciding Σ-Entailment: Theory

By Lemma 2, to decide Σ-entailment, it is sufficient to decide whether the set $\mathsf{Diff}_\Sigma(\mathcal{T}, \mathcal{T}')$ contains Σ-implications of the form $C \sqsubseteq A$ or $A \sqsubseteq D$, where A is a concept name. The latter problem is decidable in polynomial time already for general $\mathcal{EL}$-TBoxes [13]. So, in what follows we concentrate on Σ-implications of the form $C \sqsubseteq A$. We first transform $\mathcal{T}$ into a *normalised* terminology. A concept name A is called *non-conjunctive in* $\mathcal{T}$ if it is pseudo-primitive in $\mathcal{T}$ or has a definition of the form $A \equiv \exists r.C \in \mathcal{T}$. Otherwise A is called *conjunctive in* $\mathcal{T}$. A terminology $\mathcal{T}$ is normalised if it consists of axioms of the following form:

– $A \equiv \exists r.B$ or $A \sqsubseteq \exists r.B$, where B is a concept name;
– $A \equiv F$ or $A \sqsubseteq F$, where F is a (possibly empty) conjunction of concept names such that every conjunct B of F is non-conjunctive in $\mathcal{T}$.

For $A \in \mathsf{N_C}$,

- if A is pseudo-primitive in $\mathcal{T}$, then

$$\mathsf{noimply}_{\mathcal{T},\Sigma}(A) = \{\xi_A\}, \quad \mathsf{Noimply}_{\mathcal{T},\Sigma}(A) = \{\xi_A \sqsubseteq \bigsqcap_{A' \in (\Sigma \setminus \mathsf{pre}^{\Sigma}_{\mathcal{T}}(A))} A' \sqcap \mathsf{All}_\Sigma\};$$

- if A is conjunctive in $\mathcal{T}$ and $A \equiv F \in \mathcal{T}$, then

$$\mathsf{noimply}_{\mathcal{T},\Sigma}(A) = \{\xi_B \mid B \in F\}, \quad \mathsf{Noimply}_{\mathcal{T},\Sigma}(A) = \emptyset$$

- if $A \equiv \exists r.B \in \mathcal{T}$, then $\mathsf{noimply}_{\mathcal{T},\Sigma}(A) = \{\xi_A\}$ and $\mathsf{Noimply}_{\mathcal{T},\Sigma}(A) = \{\alpha_A\}$, where

$$\alpha_A = \xi_A \sqsubseteq (\bigsqcap_{A' \in (\Sigma \setminus \mathsf{pre}^{\Sigma}_{\mathcal{T}}(A))} A') \sqcap (\bigsqcap_{r \neq s \in \Sigma} \exists s.(\bigsqcap_{A' \in \Sigma} A' \sqcap \mathsf{All}_\Sigma)) \sqcap \bigsqcap_{\xi \in \mathsf{noimply}_{\mathcal{T},\Sigma}(B)} \exists r.\xi.$$

Fig. 2. Computing $\mathsf{Noimply}_{\mathcal{T},\Sigma}(A)$ and $\mathsf{noimply}_{\mathcal{T},\Sigma}(A)$

Normalised terminologies in the sense defined above are a minor modification of normalised terminologies as defined in [1]. Say that two interpretations $\mathcal{I}$ and $\mathcal{J}$ *coincide on a signature* Σ, in symbols $\mathcal{I}|_\Sigma = \mathcal{J}|_\Sigma$, if $\Delta^{\mathcal{I}} = \Delta^{\mathcal{J}}$ and $X^{\mathcal{I}} = X^{\mathcal{J}}$ for all $X \in \Sigma$.

Lemma 3. *For every terminology $\mathcal{T}$, one can construct in polynomial time a normalised terminology $\mathcal{T}'$ of polynomial size in $|\mathcal{T}|$ such that $\mathsf{sig}(\mathcal{T}) \subseteq \mathsf{sig}(\mathcal{T}')$, $\mathcal{T}' \models \mathcal{T}$, and for every model $\mathcal{I}$ of $\mathcal{T}$ there exists a model $\mathcal{J}$ of $\mathcal{T}'$ which coincides with $\mathcal{I}$ on Σ. Moreover, $\mathcal{T}'$ is acyclic if $\mathcal{T}$ is acyclic.*

The proof is a straightforward modification of the proof in [1]. From now on we will work with normalised terminologies only.

Intuitively, to decide whether there exists $C \sqsubseteq A \in \mathsf{Diff}_\Sigma(\mathcal{T},\mathcal{T}')$, we want to construct the most specific[2] Σ-concept C_A such that $\mathcal{T} \not\models C_A \sqsubseteq A$. Then there exists *some* Σ-concept C such that $C \sqsubseteq A \in \mathsf{Diff}_\Sigma(\mathcal{T},\mathcal{T}')$ if, and only if, $\mathcal{T}' \models C_A \sqsubseteq A$. Unfortunately, most specific Σ-concepts with this property do not always exist and, therefore (and also to enable structure sharing), we use an additional terminology. We use the following sets and axiom:

- $\Sigma^{\mathsf{fresh}} = \{\mathsf{All}_\Sigma\} \cup \{\xi_A \mid A \in \mathsf{N_C}$ non-conjunctive in $\mathcal{T}\}$, where All_Σ and each ξ_A are fresh concept names not occurring in $\Sigma \cup \mathsf{sig}(\mathcal{T})$;
- α denotes the concept inclusion $\mathsf{All}_\Sigma \sqsubseteq \bigsqcap_{r \in \Sigma} \exists r.(\bigsqcap_{A' \in \Sigma} A' \sqcap \mathsf{All}_\Sigma)$;
- $\mathsf{pre}^{\Sigma}_{\mathcal{T}}(A) = \{B \in \Sigma \mid \mathcal{T} \models B \sqsubseteq A\}$, for $A \in \mathsf{N_C}$. These sets can be computed in polynomial time [1].

Theorem 3. *Let $\mathcal{T}$ be a normalised terminology and Σ a signature. The terminologies $\mathsf{Noimply}_{\mathcal{T},\Sigma}(A)$ and sets of concepts names $\mathsf{noimply}_{\mathcal{T},\Sigma}(A)$ are constructed, in polynomial time, in Figure 2. Set*

[2] Recall that a concept C is more specific than a concept D if $\models C \sqsubseteq D$.

$$\mathsf{Noimply}_{\mathcal{T},\Sigma} = \{\alpha\} \cup \bigcup_{A \in \Sigma \cup \mathsf{sig}(\mathcal{T})} \mathsf{Noimply}_{\mathcal{T},\Sigma}(A).$$

The following conditions are equivalent, for every concept name $A \in \Sigma \cup \mathsf{sig}(\mathcal{T})$ and terminology $\mathcal{T}'$ with $\mathsf{sig}(\mathcal{T}') \cap \Sigma^{\mathsf{fresh}} = \emptyset$:

- *there exists a Σ-concept C with $\mathcal{T}' \models C \sqsubseteq A$ and $\mathcal{T} \not\models C \sqsubseteq A$;*
- *$\mathcal{T}' \cup \mathsf{Noimply}_{\mathcal{T},\Sigma} \models \xi \sqsubseteq A$, for some $\xi \in \mathsf{noimply}_{\mathcal{T},\Sigma}(A)$.*

Observe that, in Theorem 3, $\mathsf{Noimply}_{\mathcal{T},\Sigma}$ and $\mathsf{noimply}_{\mathcal{T},\Sigma}(A)$ do not depend on $\mathcal{T}'$. Thus, once they have been constructed, they can be used to check the existence of concept implications $C \sqsubseteq A \in \mathsf{Diff}_\Sigma(\mathcal{T},\mathcal{T}')$ for arbitrary terminologies $\mathcal{T}'$. It is worth noting as well that the proof of Theorem 3 will show that the result holds for arbitrary general TBoxes $\mathcal{T}'$ formulated in description logics which are fragments of first-order logic, and, indeed, for $\mathcal{T}'$ any first-order theory. In this case, Theorem 3 provides a reduction of checking whether there exists $C \sqsubseteq A \in \mathsf{Diff}_\Sigma(\mathcal{T},\mathcal{T}')$ to deduction in the language of $\mathcal{T}'$.

Example 2. Let $\mathcal{T} = \{A \equiv \exists r.B, B \equiv \exists r.A\}$ and $\Sigma = \{r, A, B\}$. Then we have $\mathsf{noimply}_{\mathcal{T},\Sigma}(A) = \{\xi_A\}$ and $\mathsf{Noimply}_{\mathcal{T},\Sigma} = \{\xi_A \sqsubseteq B \sqcap \exists r.\xi_B,\ \xi_B \sqsubseteq A \sqcap \exists r.\xi_A\}$. Intuitively, $\{\xi_A\} \cup \mathsf{Noimply}_{\mathcal{T},\Sigma}$ stands for the "infinitary" most specific Σ-concept not subsumed by A relative to $\mathcal{T}$.

In the remainder of this section we prove Theorem 3. To this end, we first prove an "infinitary" version of Theorem 3 by associating with every concept name A a sequence $\mathsf{noimply}^n_{\mathcal{T},\Sigma}(A)$, $n \geq 0$, of sets of Σ-concepts such that the following holds:

C1. $\mathcal{T} \not\models C \sqsubseteq A$, for all $n \geq 0$ and for all $C \in \mathsf{noimply}^n_{\mathcal{T},\Sigma}(A)$.

C2. For all Σ-concepts D, if $\mathcal{T} \not\models D \sqsubseteq A$, then $\models C \sqsubseteq D$ for some $C \in \mathsf{noimply}^n_{\mathcal{T},\Sigma}(A)$, where n is the *role-depth* $\mathsf{depth}(D)$ of D (i.e., the number of nestings of existential restrictions in D).[3]

The sets $\mathsf{noimply}^n_{\mathcal{T},\Sigma}(A)$ are defined in Figure 3. Observe that $\mathsf{noimply}^n_{\mathcal{T},\Sigma}(A)$ is well-defined because in the definition $A \equiv F \in \mathcal{T}$ of a conjunctive concept name A no conjunctive concept name occurs. This observation will also be used in the inductive proofs below.

Example 3. For the terminology $\mathcal{T}$ and signature Σ from Example 2, we have $\mathsf{noimply}^0_{\mathcal{T},\Sigma}(A) = \{B\}$, $\mathsf{noimply}^1_{\mathcal{T},\Sigma}(A) = \{B \sqcap \exists r.A\}$, $\mathsf{noimply}^2_{\mathcal{T},\Sigma}(A) = \{B \sqcap \exists r.(A \sqcap \exists r.B)\}$, etc. Thus, intuitively, $\mathsf{noimply}^n_{\mathcal{T},\Sigma}(A)$ is the unfolding up to depth n of ξ_A relative to $\mathsf{Noimply}_{\mathcal{T},\Sigma}$.

Lemma 4. *Let $\mathcal{T}$ be a normalised terminology, signature Σ, and $A \in \mathsf{N_C}$. The sets $\mathsf{noimply}^n_{\mathcal{T},\Sigma}(A)$ satisfy conditions **C1** and **C2** above.*

[3] More precisely $\mathsf{depth}(A) = 0$, $\mathsf{depth}(C_1 \sqcap C_2) = \max\{\mathsf{depth}(C_1), \mathsf{depth}(C_2)\}$, and $\mathsf{depth}(\exists r.D) = \mathsf{depth}(D) + 1$.

Set, inductively, $\mathsf{all}^0_\Sigma = \top$ and $\mathsf{all}^{n+1}_\Sigma = \bigsqcap_{r\in\Sigma} \exists r.(\bigsqcap_{A'\in\Sigma} A' \sqcap \mathsf{all}^n_\Sigma)$. Define $\mathsf{noimply}^0_{\mathcal{T},\Sigma}(A)$ as follows:

- if A is non-conjunctive in $\mathcal{T}$, then $\mathsf{noimply}^0_{\mathcal{T},\Sigma}(A) = \{\bigsqcap_{A'\in\Sigma\setminus\mathsf{pre}^\Sigma_\mathcal{T}(A)} A'\}$;
- if A is conjunctive and $A \equiv F \in \mathcal{T}$, then $\mathsf{noimply}^0_{\mathcal{T},\Sigma}(A) = \bigcup_{B\in F} \mathsf{noimply}^0_{\mathcal{T},\Sigma}(B)$;

and define, inductively, $\mathsf{noimply}^{n+1}_{\mathcal{T},\Sigma}(A)$ by

- if A is pseudo-primitive in $\mathcal{T}$, then $\mathsf{noimply}^{n+1}_{\mathcal{T},\Sigma}(A) = \{\bigsqcap_{A'\in(\Sigma\setminus\mathsf{pre}^\Sigma_\mathcal{T}(A))} A'\sqcap\mathsf{all}^{n+1}_\Sigma\}$.
- If A is conjunctive and $A \equiv F \in \mathcal{T}$, then $\mathsf{noimply}^{n+1}_{\mathcal{T},\Sigma}(A) = \bigcup_{B\in F} \mathsf{noimply}^{n+1}_{\mathcal{T},\Sigma}(B)$.
- If $A \equiv \exists r.B \in \mathcal{T}$, then $\mathsf{noimply}^{n+1}_{\mathcal{T},\Sigma}(A) = \{C^{n+1}_{\Sigma,\mathcal{T}}\}$, where

$$C^{n+1}_{\Sigma,\mathcal{T}} = (\bigsqcap_{A'\in(\Sigma\setminus\mathsf{pre}^\Sigma_\mathcal{T}(A))} A' \sqcap (\bigsqcap_{r\neq s\in\Sigma} \exists s.(\bigsqcap_{A'\in\Sigma} A' \sqcap \mathsf{all}^n_\Sigma)) \sqcap \bigsqcap_{E\in\mathsf{noimply}^n_{\mathcal{T},\Sigma}(B)} \exists r.E.$$

Fig. 3. Computing $\mathsf{noimply}^n_{\mathcal{T},\Sigma}(A)$

Proof. We start with the proof of **C1**. Assume first that A is pseudo-primitive in $\mathcal{T}$. Then $\mathsf{noimply}^n_{\mathcal{T},\Sigma}(A)$ consists of $C = \bigsqcap_{A'\in(\Sigma\setminus\mathsf{pre}^\Sigma_\mathcal{T}(A))} A' \sqcap \mathsf{all}^n_\Sigma$. By Lemma 1, $\mathcal{T} \not\models C \sqsubseteq A$ because the only atomic conjuncts of C are in $\Sigma \setminus \mathsf{pre}^\Sigma_\mathcal{T}(A)$.

We now prove **C1** for concept names A which are not pseudo-primitive in $\mathcal{T}$. The proof is by induction on n. For $n = 0$ and $A \equiv \exists r.B \in \mathcal{T}$ the claim follows again from Lemma 1 and the observation that $B' \in \mathsf{pre}^\Sigma_\mathcal{T}(A)$ if, and only if, $\mathcal{T} \models B' \sqsubseteq \exists r.B$. For $n = 0$ and A conjunctive with $A \equiv F \in \mathcal{T}$, **C1** follows since it has been proved for all conjuncts of F and $\mathcal{T} \not\models C \sqsubseteq A$ if, and only if, there exists an atomic conjunct B of F such that $\mathcal{T} \not\models C \sqsubseteq B$.

For the induction step, assume **C1** has been proved for $n \geq 0$.

Let $A \equiv \exists r.B \in \mathcal{T}$ and let $C^{n+1}_{\mathcal{T},\Sigma}$ be the only element of $\mathsf{noimply}^{n+1}_{\mathcal{T},\Sigma}(A)$. Assume $\mathcal{T} \models C^{n+1}_{\mathcal{T},\Sigma} \sqsubseteq A$. By Lemma 1 there are two cases:

- $\mathcal{T} \models \bigsqcap_{A'\in(\Sigma\setminus\mathsf{pre}^\Sigma_\mathcal{T}(A))} A' \sqsubseteq \exists r.B$. This is excluded, by Lemma 1.
- There exists $E \in \mathsf{noimply}^n_{\mathcal{T},\Sigma}(B)$ such that $\mathcal{T} \models E \sqsubseteq B$. This is excluded by the IH.

We have derived a contradiction. The case $A \equiv F \in \mathcal{T}$, A conjunctive in $\mathcal{T}$, is considered similarly to the case $n = 0$ and left to the reader.

We come to the proof of **C2**. The proof is by induction on n. Let $n = 0$ and assume first that A is non-conjunctive. Let D be a Σ-concept with $\mathsf{depth}(D) = 0$ and $\mathcal{T} \not\models D \sqsubseteq A$. Then all conjuncts of D are in $\Sigma \setminus \mathsf{pre}^\Sigma_\mathcal{T}(A)$ and we obtain $\models \bigsqcap_{A'\in\Sigma\setminus\mathsf{pre}^\Sigma_\mathcal{T}(A)} A' \sqsubseteq D$. Now assume A is conjunctive in $\mathcal{T}$ and $A \equiv F \in \mathcal{T}$. Let D be a Σ-concept with $\mathsf{depth}(D) = 0$ and $\mathcal{T} \not\models D \sqsubseteq A$. Then $\mathcal{T} \not\models D \sqsubseteq B$, for some conjunct B of F. By IH, $\models C \sqsubseteq D$ for the (unique) $C \in \mathsf{noimply}^0_{\mathcal{T},\Sigma}(B)$, and therefore $\models C \sqsubseteq D$ for some $C \in \mathsf{noimply}^0_{\mathcal{T},\Sigma}(A)$.

For the induction step, assume that **C2** has been shown for n. Let D be a Σ-concept with $\mathcal{T} \not\models D \sqsubseteq A$ and $\mathsf{depth}(D) = n + 1$. Assume first that A is pseudo-primitive in $\mathcal{T}$. Then the atomic conjuncts of D are included in $\Sigma \setminus \mathsf{pre}_{\mathcal{T}}^{\Sigma}(A)$. So, from $C = \bigsqcap_{A' \in \Sigma \setminus \mathsf{pre}_{\mathcal{T}}^{\Sigma}(A)} A' \sqcap \mathsf{all}_{\Sigma}^{n+1}$ we obtain $\models C \sqsubseteq D$.

Now assume $A \equiv \exists r.B \in \mathcal{T}$. Let $C_{\mathcal{T},\Sigma}^{n+1}$ be the only element of $\mathsf{noimply}_{\mathcal{T},\Sigma}^{n+1}(A)$ and assume

$$D = \bigsqcap_{B \in Q_0} B \sqcap \bigsqcap_{(s,D') \in Q_1} \exists s.D'.$$

Then $Q_0 \subseteq \Sigma \setminus \mathsf{pre}_{\mathcal{T}}^{\Sigma}(A)$. Hence, $\models C_{\mathcal{T},\Sigma}^{n+1} \sqsubseteq \bigsqcap_{B \in Q_0} B$. Now consider a conjunct $\exists s.D'$ of D. There are two cases:

– $s \neq r$. Then, by construction, $\models C_{\mathcal{T},\Sigma}^{n+1} \sqsubseteq \exists s.D'$.
– $s = r$. It is enough to show that there exists $E \in \mathsf{noimply}_{\mathcal{T},\Sigma}^{n}(B)$ such that $\models E \sqsubseteq D'$. Suppose there does not exist such an E. Then, by IH, $\mathcal{T} \models D' \sqsubseteq B$. Hence, $\mathcal{T} \models \exists r.D' \sqsubseteq \exists r.B$ and we obtain $\mathcal{T} \models D \sqsubseteq A$, which is a contradiction.

The case in which A is conjunctive in $\mathcal{T}$ is straightforward and is left to the reader.

Corollary 1. *For all terminologies $\mathcal{T}'$ and $A \in \mathsf{N_C}$ the following are equivalent:*

1. *there exists a Σ-concept C such that $\mathcal{T} \not\models C \sqsubseteq A$ and $\mathcal{T}' \models C \sqsubseteq A$;*
2. *there exists $n \geq 0$ and $C \in \mathsf{noimply}_{\mathcal{T},\Sigma}^{n}(A)$ such that $\mathcal{T}' \models C \sqsubseteq A$.*

Proof. The direction from Point 2 to Point 1 follows immediately from **C1**. Conversely, assume that there exists a Σ-concept C such that $\mathcal{T}' \models C \sqsubseteq A$ and $\mathcal{T} \not\models C \sqsubseteq A$. By **C1** and **C2**, there exist n and $C' \in \mathsf{noimply}_{\mathcal{T},\Sigma}^{n}(A)$ with $\models C' \sqsubseteq C$ and $\mathcal{T} \not\models C' \sqsubseteq A$. Then $\mathcal{T}' \models C' \sqsubseteq A$.

In contrast to the formulation of Theorem 3, Corollary 1 does not provide us with a polynomial time algorithm. First, no upper bound on n is given and, second, the concepts in $\mathsf{noimply}_{\mathcal{T},\Sigma}^{n}(A)$ are of exponential size in n. Example 1 is easily extended so as to show that this is unavoidable: one can construct a terminology $\mathcal{T}$ and a sequence of terminologies $\mathcal{T}_n'$ such that in minimal implications in $\mathsf{Diff}_{\Sigma}(\mathcal{T}, \mathcal{T}')$ of the form $C_n \sqsubseteq A$ the concept C_n has at least depth n and is of size 2^n. However, Theorem 3 is now an immediate consequence of the following lemma and Corollary 1.

Lemma 5. *Let $\mathcal{T}'$ be a terminology such that $\mathsf{sig}(\mathcal{T}') \cap \Sigma^{\mathsf{fresh}} = \emptyset$ and $A \in \mathsf{sig}(\mathcal{T}) \cup \Sigma$. Then the following conditions are equivalent:*

1. *$\mathcal{T}' \cup \mathsf{Noimply}_{\mathcal{T},\Sigma} \models \xi \sqsubseteq A$, for some $\xi \in \mathsf{noimply}_{\mathcal{T},\Sigma}(A)$;*
2. *$\mathcal{T}' \models C \sqsubseteq A$, for some $n \geq 0$ and $C \in \mathsf{noimply}_{\mathcal{T},\Sigma}^{n}(A)$.*

Proof. Point 2 implies Point 1. For concept names A which are non-conjunctive in $\mathcal{T}$ this follows because $\mathsf{Noimply}_{\mathcal{T},\Sigma} \models \xi_A \sqsubseteq C$ for the only element C of $\mathsf{noimply}_{\mathcal{T},\Sigma}^{n}(A)$. The conjunctive case follows by induction.

Point 1 implies Point 2 is proved by a compactness argument. Intuitively, if $\mathcal{T}' \cup \bigcup_{n \geq 0} \mathsf{noimply}_{\mathcal{T},\Sigma}^{n}(A) \not\models A$, then $\mathcal{T}' \cup \mathsf{Noimply}_{\mathcal{T},\Sigma} \not\models \xi \sqsubseteq A$, for all $\xi \in \mathsf{noimply}_{\mathcal{T},\Sigma}(A)$. However, to prove this, one has to re-construct the concepts $\mathsf{noimply}_{\mathcal{T},\Sigma}^{n}(A)$; details of the proof are given in the technical report.

5 Practical Algorithm and System

We have seen above that the sets

- $\mathsf{DiffR}_\Sigma(\mathcal{T},\mathcal{T}')$ consisting of all $A \in \Sigma$ such that there is a Σ-concept C with $\mathcal{T} \not\models C \sqsubseteq A$ and $\mathcal{T}' \models C \sqsubseteq A$, and
- $\mathsf{DiffL}_\Sigma(\mathcal{T},\mathcal{T}')$ consisting of all $A \in \Sigma$ such that there is a Σ-concept C with $\mathcal{T} \not\models A \sqsubseteq C$ and $\mathcal{T}' \models A \sqsubseteq C$

can be computed in polynomial time and can be regarded, by Lemma 2, as an informative approximation of the logical difference between $\mathcal{T}$ and $\mathcal{T}'$ w.r.t. Σ.

Computing both sets for large terminologies and signatures Σ using a direct implementation of the algorithm described above will fail: considering that state of the art description logic reasoners [2] take about 15 minutes to classify the SNOMED CT terminology [17], the reduction to reasoning given in Section 4 is impractical for large terminologies and signatures of reasonable size (the terminology $\mathsf{Noimply}_{\mathcal{T},\Sigma}$ contains huge conjunctions of Σ-concept names). We now discuss the implementation of the algorithms above in the system CEX for acyclic terminologies using a dynamic programming approach.

Let $\mathcal{T}$ and $\mathcal{T}'$ be acyclic terminologies and Σ a signature. For expositional reasons, we assume that $\Sigma \subseteq \mathsf{sig}(\mathcal{T}') \subseteq \mathsf{sig}(\mathcal{T})$. This is justified because we can add $A \sqsubseteq \top$ to $\mathcal{T}'$, for all $A \in \Sigma \setminus \mathsf{sig}(\mathcal{T}')$, and $A \sqsubseteq \top$ to $\mathcal{T}$, for all $A \in (\Sigma \cup \mathsf{sig}(\mathcal{T}')) \setminus \mathsf{sig}(\mathcal{T})$. We describe the algorithm computing DiffR_Σ, the rather straightforward algorithm computing DiffL_Σ is discussed in the technical report. We assume that $\mathcal{T}$ and $\mathcal{T}'$ are fully classified and the result of the classification is kept in a table, so, given two concept names A and B, it takes constant time to find out whether $\mathcal{T} \models A \sqsubseteq B$ (likewise, if $\mathcal{T}' \models A \sqsubseteq B$). Now the algorithm computing DiffR_Σ works by induction on concept definitions and marks, recursively, every $E \in \mathsf{sig}(\mathcal{T}')$, starting with pseudo-primitive ones, with members of $\Xi = \{\xi_A \mid A \in \mathsf{sig}(\mathcal{T}) \text{ non-conjunctive in } \mathcal{T}\}$ in such a way that

(†) $E \in \mathsf{sig}(\mathcal{T}')$ is marked with ξ if, and only if, $\mathcal{T}' \cup \mathsf{Noimply}_{\mathcal{T},\Sigma} \not\models \xi \sqsubseteq E$.

Then $A \in \Sigma$ is *not* marked with $\xi \in \mathsf{noimply}_{\mathcal{T},\Sigma}(A)$ if, and only if, $\mathcal{T}' \cup \mathsf{Noimply}_{\mathcal{T},\Sigma} \models \xi \sqsubseteq A$. If this happens to be the case for some $\xi \in \mathsf{noimply}_{\mathcal{T},\Sigma}(A)$, then A is included in $\mathsf{DiffR}_\Sigma(\mathcal{T},\mathcal{T}')$ (Theorem 3).

In order to define the marking, set $\mathsf{pre}^\Sigma_\mathcal{T}(\xi_A) = \mathsf{pre}^\Sigma_\mathcal{T}(A)$, for $A \in \mathsf{sig}(\mathcal{T})$ non-conjunctive in $\mathcal{T}$. Now mark $E \in \mathsf{sig}(\mathcal{T}')$ as follows:

1. If E is pseudo-primitive in $\mathcal{T}'$, then it is marked with all $\xi \in \Xi$ such that $\mathsf{pre}^\Sigma_{\mathcal{T}'}(E) \subseteq \mathsf{pre}^\Sigma_\mathcal{T}(\xi)$;
2. If $E \equiv E_1 \sqcap \ldots \sqcap E_k \in \mathcal{T}'$, then it is marked with all $\xi \in \Xi$ such that at least one of $E_1,\ldots, E_k$ is marked with ξ;
3. If $E \equiv \exists r.E' \in \mathcal{T}'$ and
   (a) if $r \notin \Sigma$ or $\mathcal{T}' \cup \{\alpha\} \not\models (\bigsqcap_{A' \in \Sigma} A' \sqcap \mathsf{All}_\Sigma) \sqsubseteq E'$, then E is marked with all $\xi \in \Xi$ such that $\mathsf{pre}^\Sigma_{\mathcal{T}'}(E) \subseteq \mathsf{pre}^\Sigma_\mathcal{T}(\xi)$;
   (b) if $r \in \Sigma$ and $\mathcal{T}' \cup \{\alpha\} \models (\bigsqcap_{A' \in \Sigma} A' \sqcap \mathsf{All}_\Sigma) \sqsubseteq E'$, then E is marked with all $\xi_A \in \Xi$ such that

– $A \equiv \exists r.A'$ in $\mathcal{T}$ and, for all $\xi' \in \mathsf{noimply}_{\mathcal{T},\Sigma}(A')$, E' is marked with ξ' and
– $\mathsf{pre}^{\Sigma}_{\mathcal{T}'}(E) \subseteq \mathsf{pre}^{\Sigma}_{\mathcal{T}}(\xi_A)$.

Using Theorem 3 and Lemma 5, one can prove that the defined marking has property (†). While the condition $\mathcal{T}' \cup \{\alpha\} \models (\bigsqcap_{A' \in \Sigma} A' \sqcap \mathsf{All}_\Sigma) \sqsubseteq E$ can be checked directly, this requires operating concepts of large size for large Σ's. So, instead we use the following criterion: we may assume that $\mathcal{T}$ contains a definition $A \sqsubseteq \top$ such that $A \notin \mathsf{sig}(\mathcal{T}')$ and A does not occur elsewhere in $\mathcal{T}$. Then it follows from the definitions that $\mathcal{T}' \cup \{\alpha\} \models (\bigsqcap_{A' \in \Sigma} A' \sqcap \mathsf{All}_\Sigma) \sqsubseteq E$ if, and only if, E is not marked with ξ_A.

Let T and T' be the time taken to fully classify $\mathcal{T}$ and $\mathcal{T}'$, respectively. Then all $\mathcal{T}'$ concept names can be marked in $O(|\mathcal{T}| \times |\mathcal{T}'| \times |\Sigma| + T')$ time. Overall, checking Σ-entailment takes $O(|\mathcal{T}| \times |\mathcal{T}'| \times |\Sigma| + T + T')$ time and $O(|\mathcal{T}| \times |\mathcal{T}'| \times |\Sigma|)$ space. It should be noted that in our implementation this theoretical upper bound is often not reached due to the use of hash tables and structure sharing.

6 Experimental Evaluation

CEX (see http://www.csc.liv.ac.uk/~konev/software/) is an OCaml program [4]. For the experiments, we use two versions of SNOMED CT: one dated 09 February 2005 (SM-05) and the other 30 December 2006 (SM-06) and having 379 691 and 389 472 axioms, respectively. As CEX currently accepts acyclic $\mathcal{EL}$-terminologies only, the role inclusions of SNOMED CT are not taken into account. The tests have been carried out on a standard PC: Intel® Core$^{\mathrm{TM}}$ 2 CPU at 2.13 GHz and 3 GB of RAM.

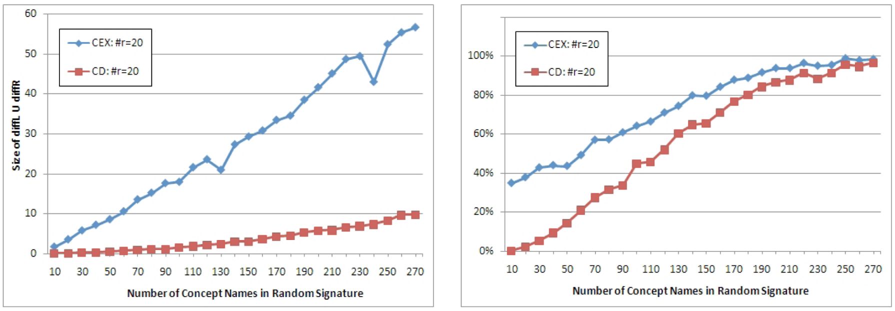
Logical difference between SM-05 *and* SM-06. Table 1 shows the average sizes of the lists $\mathsf{DiffL}_\Sigma(\text{SM-05}, \text{SM-06})$ and $\mathsf{DiffR}_\Sigma(\text{SM-05}, \text{SM-06})$ for 20 randomly generated signatures $\Sigma \subseteq \mathsf{sig}(\text{SM-05}) \cap \mathsf{sig}(\text{SM-06})$ for each of the 12 possible signature sizes containing 100, 1 000, etc. concept names and 0, 20, or 40 role names.[4] The execution time and memory consumption of CEX when computing these lists vary from 477 to 596 seconds and from 1 393 to 1 496 MByte, respectively. The numbers show that there is a huge difference between SM-05 and SM-06. Also, adding a role name to the signature has a larger impact on the number of differences than adding a concept name.

Comparison with the classification approach. We compare the size of $\mathsf{DiffL}_\Sigma \cup \mathsf{DiffR}_\Sigma$ as computed by CEX with the number of concept names in Σ for which there is a difference in the class hierarchy restricted to Σ; i.e., the set of $A \in \Sigma$ such that there exists $B \in \Sigma$ such that $A \sqsubseteq B \in \mathsf{Diff}_\Sigma$ or $B \sqsubseteq A \in \mathsf{Diff}_\Sigma$. The experiments show how many of the differences between two terminologies detected by CEX can be extracted from a straightforward comparison of class hierarchies.

[4] There are 50 role names in $\mathsf{sig}(\text{SM-05}) \cap \mathsf{sig}(\text{SM-06})$.

Table 1. Computing logical difference with CEX: Diff_Σ(SM-05, SM-06)

$\|\Sigma \cap \mathsf{N_C}\|$	$\|\Sigma \cap \mathsf{N_R}\| = 0$		$\|\Sigma \cap \mathsf{N_R}\| = 20$		$\|\Sigma \cap \mathsf{N_R}\| = 40$	
	$\|\mathsf{diffL}_\Sigma\|$	$\|\mathsf{diffR}_\Sigma\|$	$\|\mathsf{diffL}_\Sigma\|$	$\|\mathsf{diffR}_\Sigma\|$	$\|\mathsf{diffL}_\Sigma\|$	$\|\mathsf{diffR}_\Sigma\|$
100	0.10	0.10	0.90	0.15	2.95	0.20
1 000	2.35	2.15	15.55	2.95	28.85	3.75
10 000	155.35	125.35	257.35	136.20	514.10	209.90
100 000	11 795.90	4 108.60	12 954.45	4 358.30	14 942.55	6 823.60

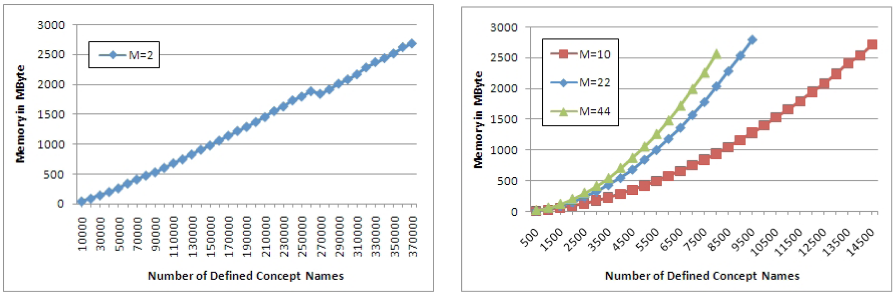


(a) Number of differences (b) Proportion of detected differences

Fig. 4. Comparison of CEX and classification-based approach

To facilitate the experiments, we use an empty terminology and an SM-05 fragment containing about 140 000 axioms. For every number between 10 and 270 with the step of 10, we generated 500 samples of a random signature containing this number of concepts and 20 roles. The results of the experiments are given in Figure 4. 4(a) shows that, for these signatures, the number of concept names CEX outputs is about five times larger than the number of concept names occurring in differences between the class hierarchies. In 4(b), we do not count the number of differences but analyse how often the two approaches detect differences at all. More precisely, we give the percentage of cases when CEX detects a difference between the two terminologies and when a difference is visible in the class hierarchies. For signatures larger than 200, both approaches almost always detect differences. But for smaller signatures there is again a significant gap between the two approaches.

Scalability. We demonstrated in the previous section that CEX is capable of finding the logical difference in two unmodified versions of SNOMED CT. In order to see how CEX's performance scales, we now test it on randomly generated acyclic terminologies of various sizes. Each randomly generated terminology contains a certain number of defined- and primitive concept names and role names. The ratio between concept equations and concept inclusions is fixed, as is the ratio between existential restrictions and conjunctions. The random terminologies



(a) Short conjunctions (b) Long conjunctions

Fig. 5. Memory consumption of CEX on randomly generated terminologies

were generated for a varying number of defined concept names using the parameters of SM-05: 62 role names; the average number of conjuncts is 2.59; the equality-inclusion ratio is 0.102; and the exists-conjunction ratio is 0.652. For every chosen size, we generate a number of samples consisting of two random terminologies as described above. We apply CEX to find the logical difference of the two terminologies over their joint signature. Figure 5 shows the time and memory consumption of CEX on randomly generated terminologies of various sizes. In 5(a) the maximum length of conjunctions was fixed as two (M=2), and in 5(b) the number of conjuncts in each conjunction is randomly selected between two and M. It can be seen that the performance of CEX crucially depends on the length of conjunctions. In 5(b), the curves break off at the point where CEX runs out of memory. For instance, in the case M=22, this happens for terminologies with more than 9 500 defined concept names.

7 Uniform Interpolation

Let $\mathcal{T}$ be a terminology and Σ a signature. A general TBox $\mathcal{T}_\Sigma$ is called a *uniform interpolant* for $\mathcal{T}$ w.r.t. Σ if $\mathsf{sig}(\mathcal{T}_\Sigma) \subseteq \Sigma$ and $\mathcal{T}_\Sigma$ and $\mathcal{T}$ are Σ-inseparable. The question whether uniform interpolants exist for every terminology $\mathcal{T}$[5] and signature Σ in a logic (i.e., whether the logic has *uniform interpolation*), has been investigated extensively in the literature, in particular in modal and intuitionistic logic [15,18,8]. For instance, modal logic K has uniform interpolation [18], but S4 does not [8]. Observe that, if a uniform interpolant $\mathcal{T}'_\Sigma$ of $\mathcal{T}'$ w.r.t. Σ exists, then $\mathcal{T}$ Σ-entails $\mathcal{T}'$ if, and only if, $\mathcal{T} \models \mathcal{T}'_\Sigma$. Thus, the problem of deciding Σ-entailment is reduced to computing a uniform interpolant and standard deduction. Unfortunately, even for $\mathcal{EL}$-terminologies uniform interpolants do not always exist.

Lemma 6. *There exists an $\mathcal{EL}$-terminology $\mathcal{T}$ and a signature Σ such that there does not exist an uniform interpolant of $\mathcal{T}$ w.r.t. Σ.*

[5] In modal or intuitionistic logic $\mathcal{T}$ is, of course, a formula.

Proof. Let $\mathcal{T} = \{A_0 \sqsubseteq B, B \sqsubseteq A_1 \sqcap \exists r.B\}$ and $\Sigma = \{A_0, A_1, r\}$. Then a uniform interpolant $\mathcal{T}_\Sigma$ would have to axiomatise (using symbols from Σ only) the class of interpretations $\mathcal{I}$ satisfying the following condition: if $d_0 \in A_0^\mathcal{I}$, then there exists a sequence $d_0 r^\mathcal{I} d_1 r^\mathcal{I} d_2 r^\mathcal{I} \dots$ with $d_i \in A_1^\mathcal{I}$ for all $i \geq 0$. It is not difficult to show that no such $\mathcal{T}_\Sigma$ exists (even in first-order logic).

On the other hand, uniform interpolants always exist for acyclic $\mathcal{EL}$-terminologies, but minimal uniform interpolants might contain exponentially many axioms.

Theorem 4. *Let $\mathcal{T}$ be an acyclic terminology and Σ a signature. Then there exists a uniform interpolant of $\mathcal{T}$ w.r.t. Σ. In the worst case, minimal uniform interpolants have exponentially many axioms.*

Proof. First, one can show that $\mathcal{T}_\Sigma = \mathcal{T}_\Sigma^l \cup \mathcal{T}_\Sigma^r$ is a uniform interpolant for $\mathcal{T}$ w.r.t. Σ if $\mathsf{sig}(\mathcal{T}_\Sigma) \subseteq \Sigma$ and

(a) $\mathcal{T} \models C \sqsubseteq A$ if, and only if, $\mathcal{T}_\Sigma^l \models C \sqsubseteq A$, for all Σ-concepts C and $A \in \Sigma$;
(b) $\mathcal{T} \models A \sqsubseteq D$ if, and only if, $\mathcal{T}_\Sigma^r \models A \sqsubseteq D$, for all Σ-concepts D and $A \in \Sigma$.

Due to space constraints we cannot describe the construction of $\mathcal{T}_\Sigma^l$ and $\mathcal{T}_\Sigma^r$ here, and refer the reader to the technical report. The following example shows that, in the worst case, minimal uniform interpolants require exponentially many axioms. Let

$$\mathcal{T} = \{A \equiv B_1 \sqcap \dots \sqcap B_n\} \cup \{A_{ij} \sqsubseteq B_i \mid 1 \leq i, j \leq n\}.$$

and $\Sigma = \{A\} \cup \{A_{ij} \mid 1 \leq i, j \leq n\}$. Then

$$\mathcal{T}_\Sigma = \{A_{1j_1} \sqcap \dots \sqcap A_{n,j_n} \sqsubseteq A \mid 1 \leq j_1, \dots, j_n \leq n\}$$

is a uniform interpolant. It is easy to see that no uniform interpolant with fewer axioms exists. This example shows as well that one has to allow for general TBoxes when constructing uniform interpolants.

The results above show that, at least from a theoretical viewpoint, deciding Σ-entailment via uniform interpolants is less efficient than the approach discussed before. Still, uniform interpolants are useful for a number of applications, and it would be of interest to see whether this approach is viable for real-world terminologies.

8 Discussion

We have shown that computing the logical difference is tractable for $\mathcal{EL}$-terminologies and that this approach exhibits differences which are not visible in the class hierarchy. Our experiments with SNOMED CT show that the algorithm can be implemented in such a way that very large terminologies can be compared efficiently.

The following result shows that there is no straightforward way of extending these results to (even acyclic) terminologies in the basic Boolean description logic $\mathcal{ALC}$ (in which concepts can be constructed using, in addition, negation).

Theorem 5. *(1) Σ-entailment is* NExpTime*-hard for acyclic $\mathcal{ALC}$-terminologies. (2) Uniform interpolants do not always exist for acyclic $\mathcal{ALC}$-terminologies.*

Proof. Point (1) can be proved by a reduction of the NExpTime-hard problem of deciding conservative extensions in modal logic K [7], details are given in the technical report.

Point (2). We rewrite the terminology from Lemma 6. Let $\mathcal{T} = \{A \sqsubseteq (\neg A_0 \sqcup B) \sqcap (\neg B \sqcup (A_1 \sqcap \exists r.B))\}$ and $\Sigma = \{A, A_0, A_1, r\}$. It follows from the proof of Lemma 6 that there does not exist a general $\mathcal{ALC}$-TBox $\mathcal{T}_\Sigma^A$ axiomatising (using only the symbols from Σ) the class $\mathcal{S}$ of interpretations $\mathcal{I}$ satisfying the following conditions: $A^\mathcal{I} = \Delta^\mathcal{I}$ and if $d_0 \in A_0^\mathcal{I}$, then there exists a sequence $d_0 r^\mathcal{I} d_1 r^\mathcal{I} d_2 r^\mathcal{I} \ldots$ with $d_i \in A_1^\mathcal{I}$ for all $i \geq 0$. Now assume that there exists a uniform interpolant $\mathcal{T}_\Sigma$ of $\mathcal{T}$ w.r.t. Σ. Then $\mathcal{T}_\Sigma \cup \{A \equiv \top\}$ would be an axiomatisation of $\mathcal{S}$ and we have derived a contradiction.

Point (2) of Theorem 5 is slightly unexpected, because it shows that it is not possible to lift results from modal logic K (which has uniform interpolation) to acyclic $\mathcal{ALC}$-terminologies. Besides of considering extensions of our approach to languages with additional concept constructors, such as $\mathcal{ALC}$, directions for future research include terminologies with additional role boxes. Snomed ct has an additional role box consisting of implications $r \sqsubseteq r'$, $r \circ s \sqsubseteq r$ (right-identities), and $s \circ r \sqsubseteq r$ (left-identities), where r, s, r' are role names. It is not difficult to extend the algorithm (and implementation) presented in this paper to terminologies containing implications of the first type, but it remains open whether Σ-entailment is still tractable for additional role boxes containing left- and right-identities.

Finally, for the system CEX to be useful in practice, the outputs DiffL_Σ and DiffR_Σ have to be expanded by suggesting, for $A \in \mathsf{DiffR}_\Sigma$, Σ-concepts C such that $C \sqsubseteq A \in \mathsf{Diff}_\Sigma$, and similarly for DiffL_Σ. Computing such C's is straightforward by unfolding the concept ξ_A relative to $\mathsf{Noimply}_{\mathcal{T},\Sigma}$. However, even this might not provide enough information, because for the user it could be difficult to find out which difference between the axioms of the two terminologies has caused a certain Σ-difference. Thus, as a second step one might consider pinpointing algorithms *explaining* from which axioms of a terminology a counterexample $C \sqsubseteq A$ is derivable [3].

Acknowledgements. The authors were supported by EPSRC grant EP/E065279/1.

References

1. Baader, F.: Terminological cycles in a description logic with existential restrictions. In: Proceedings of IJCAI 2003, pp. 325–330. Morgan Kaufmann, San Francisco (2003); Long version available as LTCS Report 02-02
2. Baader, F., Lutz, C., Suntisrivaraporn, B.: CEL—a polynomial-time reasoner for life science ontologies. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 287–291. Springer, Heidelberg (2006)

3. Baader, F., Peñaloza, R., Suntisrivaraporn, B.: Pinpointing in the description logic $\mathcal{EL}^+$. In: Hertzberg, J., Beetz, M., Englert, R. (eds.) KI 2007. LNCS (LNAI), vol. 4667, pp. 52–67. Springer, Heidelberg (2007)
4. The Caml team, http://caml.inria.fr/contact.en.html
5. Flögel, A., Büning, H.K., Lettmann, T.: On the restricted equivalence of subclasses of propositional logic. ITA 27(4), 327–340 (1993)
6. Ghilardi, S., Lutz, C., Wolter, F.: Did I damage my ontology? a case for conservative extensions in description logics. In: Proceedings of KR 2006, pp. 187–197. AAAI Press, Menlo Park (2006)
7. Ghilardi, S., Lutz, C., Wolter, F., Zakharyaschev, M.: Conservative extensions in modal logics. In: Proceedings of AiML-6, pp. 187–207. College Publications (2006)
8. Ghilardi, S., Zawadowski, M.: Undefinability of propositional quantifiers in the modal system S4. Studia Logica 55(2), 259–271 (1995)
9. Grau, B.C., Horrocks, I., Kazakov, Y., Sattler, U.: Just the right amount: extracting modules from ontologies. In: Proceedings of WWW 2007, pp. 717–726. ACM Press, New York (2007)
10. Hofmann, M.: Proof-theoretic approach to description logic. In: Proceedings of LICS 2005, pp. 229–237. IEEE Computer Society Press, Los Alamitos (2005)
11. Konev, B., Walther, D., Wolter, F.: The logical difference problem for description logic terminologies (manuscript 2008), http://www.csc.liv.ac.uk/~frank/publ/publ.html
12. Lutz, C., Walther, D., Wolter, F.: Conservative extensions in expressive description logics. In: Proceedings of IJCAI 2007, pp. 453–458. AAAI Press, Menlo Park (2007)
13. Lutz, C., Wolter, F.: Conservative extensions in the lightweight description logic $\mathcal{EL}$. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 84–99. Springer, Heidelberg (2007)
14. Noy, N.F., Musen, M.: Promptdiff: A fixed-point algorithm for comparing ontology versions. In: Proceedings of AAAI 2002, pp. 744–750. AAAI Press, Menlo Park (2002)
15. Pitts, A.: On an interpretation of second-order quantification in first-order intuitionistic propositional logic. Journal of Symbolic Logic 57(1), 33–52 (1992)
16. Sofronie-Stokkermans, V.: Interpolation in local theory extensions. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 235–250. Springer, Heidelberg (2006)
17. Spackman, K.: Managing clinical terminology hierarchies using algorithmic calculation of subsumption: Experience with SNOMED-RT. In: JAMIA, Fall Symposium Special Issue (2000)
18. Visser, A.: Uniform interpolation and layered bisimulation. In: Gödel 1996 (Brno, 1996). Lecture Notes Logic, vol. 6, pp. 139–164. Springer, Heidelberg (1996)

Aligator: A Mathematica Package for Invariant Generation (System Description)

Laura Kovács*

EPFL, Switzerland
laura.kovacs@epfl.ch

Abstract. We describe the new software package `Aligator` for automatically inferring polynomial loop invariants. The package combines algorithms from symbolic summation and polynomial algebra with computational logic, and is applicable to the rich class of P-solvable loops. `Aligator` contains routines for checking the P-solvability of loops, transforming them into a system of recurrence equations, solving recurrences and deriving closed forms of loop variables, computing the ideal of polynomial invariants by variable elimination, invariant filtering and completeness check of the resulting set of invariants.

1 Introduction

In [6,7] we defined a family of loops, called *P-solvable*, with assignments, sequencing and conditionals, and ignored test conditions. For these loops the value of each program variable in a loop iteration can be expressed as a polynomial of the initial values of variables, the loop counter, and some new variables where there are algebraic dependencies among the new variables. For such loops, a systematic method is developed for generating a set of polynomial equations as invariants. We call a *polynomial equation* or *polynomial identity* any equation $p = 0$ where p is a polynomial and a *polynomial invariant* any loop invariant that is a conjunction of polynomial equations.

The approach described in [6,7] can be summarized as follows. (i) P-solvable loops with nested conditionals are first transformed into a nested loop system with assignments only. (ii) Each P-solvable loop with assignments only is written as a system of recurrence equations in the loop counter. (iii) Recurrences are then solved exactly by symbolic summation algorithms [3,12], and algebraic dependencies among possible exponential sequences occurring in the closed forms are derived using polynomial algebra and algorithmic combinatorics [4]. (iv) Next, the loop counter and the variables standing for the exponential sequences are eliminated by a Gröbner basis computation [1]. A finite set of polynomial identities among the program variables as invariants is thus obtained, from which, under additional conditions for loops with conditionals, *all* polynomial invariants of P-solvable loops can be inferred.

* Research was partly done while the author was at the Research Institute for Symbolic Computation (RISC), Linz, Austria.

Many non-trivial algorithms working on numbers can be naturally implemented using P-solvable loops.

The purpose of this paper is to give an overview of a new software package, called `Aligator`, for generating automatically polynomial invariants of P-solvable loops. General principles of the implementation will be discussed and illustrated on motivating examples. For the soundness of our approach we refer to [6,7].

`Aligator` was implemented in *Mathematica* 5.2 [11], and is available from:

`http://mtc.epfl.ch/software-tools/Aligator/`

`Aligator` includes algorithms solving special classes of recurrence relations (those that are either Gosper-summable [3] or C-finite [12]), generating polynomial dependencies among algebraic exponential sequences [4], and performing variable elimination using Gröbner basis computation [1]. The current version slightly differs from the one given in [6] due to the integration of our implementation of C-finite recurrence solving in `Aligator`.

We begin with loading the package into *Mathematica*, as given below.

```
In[1]:= << Aligator.m
        Automated Loop Invariant Generation by Algebraic Techniques Over the Rationals.
        Package written by Laura Kovacs - ©RISC Linz and EPFL Lausanne - V 0.3 (2008-02-01)
```

The examples presented in the paper can be directly fed into *Mathematica*, after loading `Aligator`; only minor changes in output lines were made to improve readability. We have successfully tried our implementation on many examples, see [6] or the mentioned `URL`; for each of the examples results were obtained in less than 3.5 seconds on a machine with 2.0GHz CPU and 2GB of memory.

2 `Aligator` - Underlying Principles

`Aligator` commands have syntax and semantics similar to those of *Mathematica*. However, in order to avoid any conflict with *Mathematica*'s `While` and `If` commands we use the symbols `WHILE` and `IF` for denoting loops and conditionals. The syntax of considered P-solvable loops is as follows.

$$\texttt{WHILE}[b_0, s_0; \texttt{IF}[b_1, s_1, \texttt{IF}[b_2, \ldots\ldots, \texttt{IF}[b_{k-1}, s_{k-1}, s_k]\ldots]]; s_{k+1}], \qquad (1)$$

where $s_0, \ldots, s_{k+1}$ are sequences of assignments and $b_0, \ldots, b_{k-1}$ are boolean expressions.

In our approach to invariant generation tests are ignored, and we thus deal with (basic) non-deterministic programs [6]. We omit the condition b from $\texttt{IF}[b,\ S_1,\ S_2]$, where S_1 (then-branch) and S_2 (else-branch) are sequences of assignments, and write it as $\texttt{IF}[\ldots,\ S_1,\ S_2]$ to mean the non-deterministic program $S_1|S_2$. Similarly, we omit the condition b from $\texttt{WHILE}[b, S]$, where S is a sequence of assignments, and write it in the form $\texttt{WHILE}[\ldots, S]$ to mean the non-deterministic program S^*. (1) is thus treated internally as

$$(S_1|\ldots|S_k)^*, \text{ where } S_i = s_0; s_i; s_{k+1}. \qquad (2)$$

In the sequel, let X denote the set of loop variables, and X_0 the corresponding initial values (before entering the loop). To invoke `Aligator` one uses the following command.

Command 2.1 : `Aligator[PLoop, IniVal→list of assignments]`

Input: a P-solvable loop `PLoop` as in (1), and *optionally* a `list of assignments` specifying the initial values X_0 of X

Output: a `completeness message` and
a polynomial loop invariant $p_1(X) = 0 \wedge \cdots \wedge p_r(X) = 0$

where the `completeness message` is either "Method is complete!" or "Cannot determine completeness of the method!", and refers to the fact whether the computed set of polynomials is a basis of the polynomial invariant ideal (see page 281 for more details).

EXAMPLE 2.1 (An algorithm computing the closest integer r to the cubic root of integer a [5,9].)

In[2]:= `Aligator[WHILE[`$x - s > 0, x := x - s; s := s + 6 * r + 3; r := r + 1$`],`
`IniVal→`$\{x := a; r := 1; s := 13/4\}$`]`

```
Method is complete!
```

Out[2]:= $\frac{1}{4} + 3r^2 = s \ \wedge \ 1 + 4a + 6r^2 = 3r + 4r^3 + 4x$

EXAMPLE 2.2 (Wensley's algorithm for real division [10,9].)

In[3]:= `Aligator[WHILE[`$d \geq Tol,$ `IF[`$P < a + b,$
$b := b/2; d := d/2,$
$a := a + b; y := y + d/2; b := b/2; d := d/2$`]],`
`IniVal→` $\{a := 0; b := Q/2; d := 1; y := 0\}$`]`

```
Method is complete!
```

Out[3]:= $2b = dQ \ \wedge \ ad = 2by \ \wedge \ a = Qy$

Internally, `Aligator` first checks whether a given input is as (1). If it is not, an error is reported, and the execution stops. Otherwise, polynomial invariants are generated by calling (i) the routine `InvLoopAssg` if the loop has only assignments, or (ii) `InvLoopCond` in case the loop has conditionals. We briefly discuss the main steps of these routines in Section 3, and illustrate how they are connected in Figure 1.

3 Inside `Aligator` - Main Steps of the Tool

Loop Transformation. Transforming loops as on page 276 is achieved by the command `IfWhileTransform` presented below. In this command we use a construct `Body` and write `Body[S]` in order to maintain the loop body `S` in an unevaluated form.

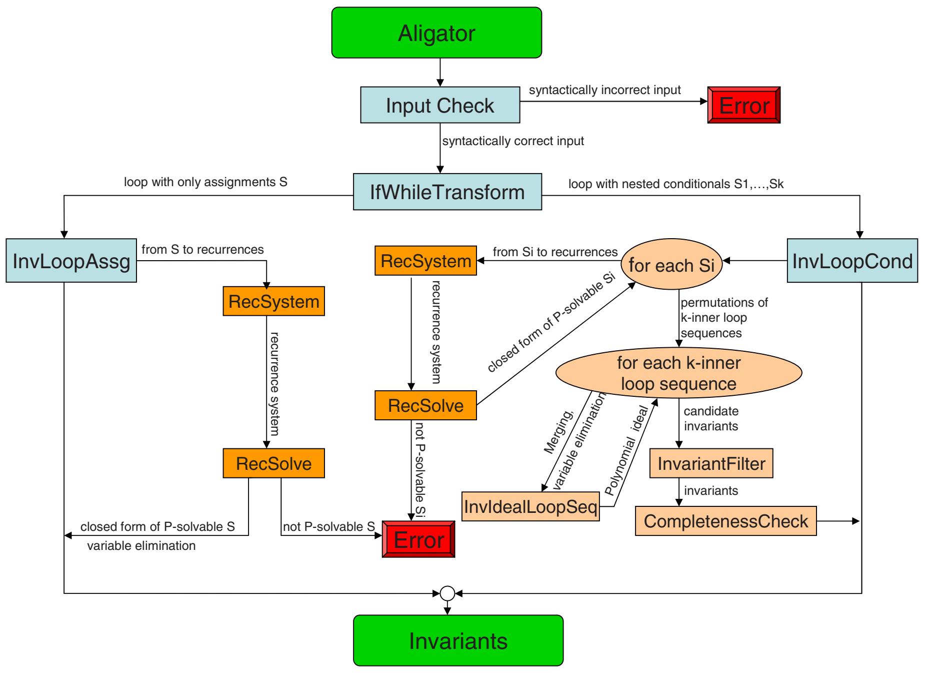


Fig. 1. The `Aligator` tool

Command 3.1 : `IfWhileTransform [PLoop, Body[], Body[]]`

Input: P-solvable loop `PLoop` as in (1)

Output: List of inner loops $\{$Body$[S_1]$,...,Body$[S_k]\}$ cf. notations from (2)

EXAMPLE 3.1 For Example 2.1 we proceed as below.

In[4]:= IfWhileTransform[WHILE$[x - s > 0, x := x - s; s := s + 6 * r + 3;$
$r := r + 1]$, Body[], Body[]]

Out[4]:= Body$[x := x - s; s := s + 6r + 3; r := r + 1]$

EXAMPLE 3.2 For Example 2.2 we obtain:

In[5]:= IfWhileTransform[WHILE$[d \geq Tol$, IF$[P < a + b,\ b := b/2; d := d/2,$
$a := a + b; y := y + d/2; b := b/2; d := d/2]]$,
Body[], Body[]]

Out[5]:= $\{$Body$[b := b/2; d := d/2]$, Body$[a := a + b; y := y + d/2; b := b/2; d := d/2]\}$

Loops and Recurrences. The recurrence equations are constructed by `RecSystem`:

Command 3.2 : `RecSystem[`S_i`]`

Input: an assignment sequence S_i corresponding to the body of the ith inner loop

Output: {`recEqs`, `varList`, {`recVar`}}, where:

- `recEqs` is the system of recurrence equations corresponding to (*flattened*) S_i;
- `varList` is the list of correspondences between sequences and variables;
- and `recVar` is a fresh variable denoting the iteration counter of the ith inner loop.

EXAMPLE 3.3 The recurrence equations of the loop from Example 2.1 are as follows.

In[6]:= `RecSystem`$[x := x - s; s := s + 6 * r + 3; r := r + 1]$

Out[6]:= $\{\{x[1+n] = -s[n] + x[n], s[1+n] = 3 + 6r[n] + s[n], r[1+n] = 1 + r[n]\},$
$\{\{x[n], x\}, \{s[n], s\}, \{r[n], r\}\}, \{n\}\},$

where n is the loop counter.

EXAMPLE 3.4 The recurrence equations of the second inner loop of Example 2.2 are:

In[7]:= `RecSystem`$[a := a + b; y := y + d/2; b := b/2; d := d/2]$

Out[7]:= $\Big\{\{a[1+m] = a[m] + b[m], y[1+m] = d[m]/2 + y[m], b[1+m] = b[m]/2,$
$d[1+m] = d[m]/2\},\ \{\{a[m], a\}, \{y[m], y\}, \{b[m], b\}, \{d[m], d\}\},\ \{m\}\Big\},$

where m is the loop counter.

Solving recurrences and checking P-solvability of a loop is embedded in the `RecSolve` routine. `RecSolve` includes packages for solving Gosper-summable recurrences [8], and for deriving algebraic dependencies among exponential sequences [4]. Contrarily to [6], for solving C-finite recurrences we now rely on our own implementation.

Command 3.3 : `RecSolve[recEqs, varList, {recVar}]`

Input: {`recEqs`, `varList`, {`recVar`}} is the output of `RecSystem`

Output: {`CFSystem`, {`recVar`}, `expVars`, `finVars`, `iniVars`, `algDep`}, where

- `CFSystem` is the polynomial closed form system of `recSystem`, and `finVars` is the list of variables whose closed forms are computed with initial values `iniVars`;
- `expVars` is the list of fresh variables denoting the exponential sequences from `CFSystem`, and `algDep` is a basis of the algebraic dependencies among `expVars`.

If the loop with recurrence system `recEqs` is not P-solvable, `RecSolve` reports an error, and execution is aborted.

EXAMPLE 3.5 Using Example 3.3, we derive the closed form system of Example 2.1.

In[8]:= `RecSolve`$[\{x[1+n] = x[n] - s[n], s[1+n] = s[n] + 6r[n] + 3, r[1+n] = r[n] + 1\},$
$\{\{x[n], x\}, \{s[n], s\}, \{r[n], r\}\}, \{n\}\ ]$

Out[8]:= $\Big\{\Big\{x = x[0] - \frac{1}{2}n(1 + 2n^2 - 6r[0] + n(-3 + 6r[0]) + 2s[0]), s = s[0] + 3n(n + 2r[0]),$
$r = r[0] + n\Big\},\ \{n\},\ \{\},\ \{x, s, r\},\ \{x[0], s[0], r[0]\},\ \{\}\Big\}$

EXAMPLE 3.6 The closed form system of the 2nd inner loop of Example 2.1 is:

In[9]:= `RecSolve`$[\{a[1+m]=a[m]+b[m], y[1+m]=d[m]/2+y[m], b[1+m]=b[m]/2,$
$d[1+m]=d[m]/2\}, \{\{a[m],a\},\{y[m],y\},\{b[m],b\},\{d[m],d\}\},\{m\}]$

Out[9]:= $$\Big\{\{a=a[0]+2b[0]-2x_3b[0],\ y=d[0]-x_5d[0]+y[0],\ b=x_1b[0],\ d=x_2d[0]\},$$
$$\{m\},\ \{x_1,x_2,x_3,x_4,x_5\},\ \{a,y,b,d\},\ \{a[0],y[0],b[0],d[0]\},$$
$$\{x_1-x_5=0,\ x_2-x_5=0,\ x_3-x_5=0,\ -1+x_4x_5=0\}\Big\}$$

where $x_1=x_2=x_3=x_5=2^{-m}$ and $x_4=2^m$.

Invariant Generation for P-solvable Loops with Assignments Only. We have now all ingredients to synthesize our invariant generation algorithm for *P-solvable loops with assignments only*, implemented in the `InvLoopAssg` routine. The main steps are as follows. (a) The loop body is modeled by recurrence equations; (b) Polynomial closed forms of recurrences are derived; (c) The loop counter and new variables denoting exponential sequences are eliminated by Gröbner basis computation from the closed form system, using the algebraic dependencies among the new variables. The derived set of polynomials forms a *basis of the polynomial invariant ideal* [6].

Command 3.4 : `InvLoopAssg[Body[S]]`

Input: a P-solvable loop body S, where S is a sequence of assignments

Output: the "Method is complete!" message and a polynomial invariant ideal basis $\{p_1,\dots,p_r\}$

The output of `Aligator` is thus obtained by taking the conjunction of the polynomial equalities $p_1=0\wedge\cdots\wedge p_r=0$. If a P-solvable loop has no polynomial invariant, `Aligator` returns `True` corresponding to the polynomial basis $\{\}$.

EXAMPLE 3.7 For Example 2.1, the internal execution flow of `InvLoopAssg` is the chain Example 3.3 → Example 3.5. From Example 3.1, we thus have:

In[10]:= `InvLoopAssg[Body[`$x:=x-s; s:=s+6r+3; r:=r+1$`]]`

Method is complete!

Out[10]:= $$\{3r^2-s-3r[0]^2+s[0],$$
$$r-3r^2+2r^3+2x-r[0]+3r[0]^2-6rr[0]^2+4r[0]^3+2rs[0]-2r[0]s[0]-2x[0]\}$$

where $r[0], s[0], x[0]$ are respectively the initial values of r, s, x.

Invariant Generation for P-solvable Loops with (Nested) Conditionals. Loop (2) is P-solvable if its inner loops $S_1,\dots,S_k$ are P-solvable. `InvLoopAssg` is thus a special case ($k=1$) of `InvLoopCond`. The main steps of `InvLoopCond` for generating invariants of *P-solvable loops with assignments and conditionals* are given below.

(a) Closed forms of S_i are first derived as discussed above, see e.g. Examples 3.4 and 3.6. Next, (b) closed forms of inner loops are taken in all $k!$ possible permutations of the k inner loop sequences. Hence, (c) the ideal of polynomial relations of a

k-sequence of inner loops is derived by eliminating inner loop counters and new variables denoting exponential sequences from the *polynomial merged closed form system* of the inner loops, using the algebraic dependencies among the new variables. This is performed by the command `InvIdealLoopSeq` - for details, we refer to the `URL` of `Aligator`. Further, (d) by taking the intersection of all $k!$ computed ideals, we derive the ideal I of valid polynomial relations for any k-sequence with initial values X_0. That is the ideal of polynomial relations after an iteration of (2), with initial values X_0. (e) From I, we only keep those polynomials that are preserved by each S_i, in the sense of *weakest precondition* `wp` *computation* [2]. Namely, we compute the set of polynomials $GI = \{p \in I \mid \mathtt{wp}(S_i, p(X) = 0) \in \langle GI \rangle,\ i = 1, \dots, k\}$ from I that yield polynomial invariants [6,7]. This is performed by the `InvariantFilter` command.

Command 3.5 : `InvariantFilter[I, rules, X]`

Input: $\{\mathtt{I}, \mathtt{rules}, \mathtt{X}\}$, where `rules` is the list of rewrite rules corresponding to the inner loop bodies

Output: $\mathtt{GI} \subset \mathtt{I}$

EXAMPLE 3.8 For Example 2.2, we have

$$\mathtt{I} = \{bd[0] - db[0], ad[0] - a[0]d[0] + 2b[0]y[0] - 2yb[0], ad - 2by - da[0] + 2by[0]\}$$
and $\mathtt{rules} = \{\{b \to b/2, d \to d/2\}, \{a \to a+b, y \to y + d/2, b \to b/2, d \to d/2\}\}$.

The result of filtering is:

In[11]:= $\mathtt{InvariantFilter[I, rules,} \{a, y, b, d\}]$

Out[11]:= $\{-db[0] + bd[0], -2yb[0] + ad[0] - a[0]d[0] + 2b[0]y[0], ad - 2by - da[0] + 2by[0]\}$

Finally, (f) GI is checked for *completeness*, based on ideal theoretic operations. Completeness is established if (i) $\langle GI \rangle = I$; or (ii)$\langle GI \rangle = I \cap I'$, with I' being the ideal of polynomial relations after all $k+1$ inner loop sequences with X_0 initial values. Such an inner loop sequence of length $k+1$ is obtained by taking a permutation of k inner loops, i.e. $S^*_{j_1}; \dots; S^*_{j_k}$, followed by an inner loop S_i with $i \neq j_k$. For details we refer to [6,7].

For all examples we found, `Aligator` returns a basis of the polynomial invariant ideal. We thus conjecture that our approach is complete for a rich class of P-solvable loops.

Command 3.6 : `InvLoopCond[`$\{$`Body[`S_1`], ..., Body[`S_k`]`$\}$`]`

Input: P-solvable inner loop bodies $S_1, \dots, S_k$, where S_i are sequences of assignments

Output: "completeness message" and $\{p_1, \dots, p_r\}$ s. t. $p_1 = 0 \wedge \dots \wedge p_r = 0$ is invariant

EXAMPLE 3.9 For Example 2.2, we derive:

In[12]:= $\texttt{InvLoopCond}[\{\texttt{Body}[b := b/2; d := d/2],$
$\texttt{Body}[a := a + b; y := y + d/2; b := b/2; d := d/2]\}]$

Method is complete!

Out[12]:= $\{-db[0] + bd[0], -2yb[0] + ad[0] - a[0]d[0] + 2b[0]y[0], ad - 2by - da[0] + 2by[0]\}$

where $a[0], b[0], d[0], y[0]$ are respectively the initial values of a, b, d, y.

4 Conclusions

`Aligator` offers software support for **a**utomated **i**nvariant **g**eneration by **a**lgebraic **t**echniques **o**ver the **r**ationals. The successful application of the package on a number of examples demonstrates the value of using symbolic summation and polynomial algebra together with computational logic for program verification.

References

1. 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)
2. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, Englewood Cliffs (1976)
3. Gosper, R.W.: Decision Procedures for Indefinite Hypergeometric Summation. J. of Symbolic Computation 75, 40–42 (1978)
4. Kauers, M., Zimmermann, B.: Computing the Algebraic Relations of C-finite Sequences and Multisequences. J. of Symbolic Computation (to appear, 2007)
5. Knuth, D.E.: The Art of Computer Programming. In: Seminumerical Algorithms, 3rd edn., vol. 2. Addison-Wesley, Reading (1998)
6. Kovács, L.: Automated Invariant Generation by Algebraic Techniques for Imperative Program Verification in Theorema. PhD thesis, RISC, Johannes Kepler University Linz (2007)
7. Kovács, L.: Reasoning Algebraically About P-Solvable Loops. In: Proc. of TACAS, Budapest, Hungary. LNCS, vol. 4963, pp. 249–264. Springer, Heidelberg (to appear, 2008)
8. Paule, P., Schorn, M.: A Mathematica Version of Zeilberger's Algorithm for Proving Binomial Coefficient Identities. J. of Symbolic Computation 20(5-6), 673–698 (1995)
9. Rodriguez-Carbonell, E., Kapur, D.: Generating All Polynomial Invariants in Simple Loops. J. of Symbolic Computation 42(4), 443–476 (2007)
10. Wegbreit, B.: The Synthesis of Loop Predicates. Comm. of the ACM 2(17), 102–112 (1974)
11. Wolfram, S.: The Mathematica Book. Version 5.0. Wolfram Media (2003)
12. Zeilberger, D.: A Holonomic System Approach to Special Functions. J. of Computational and Applied Mathematics 32, 321–368 (1990)

leanCoP 2.0 and ileanCoP 1.2: High Performance Lean Theorem Proving in Classical and Intuitionistic Logic (System Descriptions)

Jens Otten

Institut für Informatik, University of Potsdam
August-Bebel-Str. 89, 14482 Potsdam-Babelsberg, Germany
jeotten@cs.uni-potsdam.de

Abstract. leanCoP is a very compact theorem prover for classical first-order logic, based on the connection (tableau) calculus and implemented in Prolog. leanCoP 2.0 enhances leanCoP 1.0 by adding regularity, lemmata, and a technique for restricting backtracking. It also provides a definitional translation into clausal form and integrates "Prolog technology" into a lean theorem prover. ileanCoP is a compact theorem prover for intuitionistic first-order logic and based on the clausal connection calculus for intuitionistic logic. ileanCoP 1.2 extends the classical prover leanCoP 2.0 by adding prefixes and a prefix unification algorithm. We present details of both implementations and evaluate their performance.

1 Introduction

Connection calculi, such as the connection calculus [3,4], the connection tableau calculus [8,9], and the model elimination calculus [10], are well known for their goal-oriented proof search. Several implementations that are based on these calculi have been developed, for example KoMeT [5], METEOR [1], PTTP [22], SETHEO [7], and leanCoP [16].

leanCoP is an automated theorem prover for classical first-order logic. It is a very compact Prolog implementation of the connection calculus. leanCoP 2.0 enhances leanCoP 1.0 [16] by adding regularity, lemmata, and a technique to restrict backtracking [15]. In contrast to leanCoP 1.0, the input clauses are stored in Prolog's, database, which makes it possible to use Prolog's built-in indexing mechanism. Furthermore leanCoP 2.0 provides a definitional translation into clausal form and uses a fixed strategy scheduling.

ileanCoP is an automated theorem prover for intuitionistic first-order logic and is based on the clausal connection calculus for intuitionistic logic [14]. It extends leanCoP by adding a prefix to each literal and a prefix unification algorithm [17]. ileanCoP 1.2 enhances ileanCoP 1.0 by integrating the new inference rules and techniques of leanCoP 2.0.

Details of the architecture and the implementation of both leanCoP 2.0 and ileanCoP 1.2 are presented in Section 2 and Section 3, respectively. We also present performance results on the TPTP library and the MPTP challenge.

2 leanCoP 2.0 for Classical Logic

We first describe the new search techniques of leanCoP 2.0 and provide details of the source code, before presenting some performance results.

2.1 Architecture

leanCoP [16] is based on the clausal connection (tableau) calculus [4,9]. A derivation for a formula in clausal form is generated by first applying the start rule and then repeatedly applying the reduction or the extension rule. In each inference step a connection is identified along an active path. Iterative deepening on the length of the active path is performed. The following techniques are the main improvements of leanCoP 2.0 compared to leanCoP 1.0 [16]:

1. *Lean Prolog technology*: For all input clauses C and $\texttt{L}{\in}C$ of the given formula the fact `lit(L,C1,Grnd)` is stored in Prolog's database, where $\texttt{C1}{=}C\backslash\{\texttt{L}\}$ is a list and `Grnd` is `g` iff C is ground and `n` otherwise. Atoms are represented by Prolog atoms, negation is represented by "`-`". This new technique integrates the main advantage of the "Prolog technology" approach [22] by using Prolog's fast indexing mechanism to quickly find connections [15].
2. *Controlled iterative deepening*: Iterative deepening is aborted if the current limit for the length of the active path is not exceeded during the proof search.
3. *Definitional clausal form translation*: Formulae that are not in clausal form are translated into clausal form by introducing definitions for certain subformulae. In contrast to approaches for saturation-based calculi, our definitional translation is designed to work well with connection calculi [15].
4. *Regularity*: The proof search is restricted to proofs where no literal occurs more than once in the (currently investigated) active path [9].
5. *Lemmata*: A branch with a literal L that has already been closed can be reused to close other branches (below/to the right of L) that contain L [9].
6. *Restricted backtracking*: Once the application of the reduction or extension rule has successfully closed a branch by an appropriate connection, all alternative connections are cut off (on backtracking). Furthermore, alternative start clauses of the start rule are cut off (on backtracking). This new technique improves performance significantly, in particular for formulae containing many axioms, e.g. equality axioms [15].
7. *Strategy scheduling*: A list of settings is used to control the proof search (see below). The core Prolog prover is consecutively invoked by a shell script with different settings.

The minimal source code of the core prover is shown in Figure 1. It is invoked with `prove(1,S)` where `S` is a list of settings. The predicate succeeds iff there is a connection proof for the clauses stored in Prolog's database.[1] The full source code of the core prover and of the definitional clausal form translation is available on the leanCoP website. A detailed explanation of the source code, the underlying calculus and the new proof search techniques can be found in [15].

[1] Sound unification has to be switched on. In ECLiPSe Prolog this is done with `set_flag(occur_check,on)`.

```
prove(I,S) :- \+member(scut,S) -> prove([-(#)],[],I,[],S) ;
    lit(#,C,_) -> prove(C,[-(#)],I,[],S).
prove(I,S) :- member(comp(L),S), I=L -> prove(1,[]) ;
    (member(comp(_),S);retract(p)) -> J is I+1, prove(J,S).
prove([],_,_,_,_).
prove([L|C],P,I,Q,S) :- \+ (member(A,[L|C]), member(B,P),
    A==B), (-N=L;-L=N) -> ( member(D,Q), L==D ;
    member(E,P), unify_with_occurs_check(E,N) ; lit(N,F,H),
    (H=g -> true ; length(P,K), K<I -> true ;
    \+p -> assert(p), fail), prove(F,[L|P],I,Q,S) ),
    (member(cut,S) -> ! ; true), prove(C,P,I,[L|Q],S).
```

Fig. 1. The source code of the leanCoP 2.0 core prover

The list S of settings can contain one or more of the following options:

1. nodef/def: The standard/definitional translation into clausal form is done. If none of these options is given the standard translation is used for the axioms whereas the definitional translation is used for the conjecture.
2. conj: The special literal # is added to the conjecture clauses in order to mark them as possible start clauses. If this option is not given the literal # is added to all positive clauses to mark them as possible start clauses.
3. reo(I): After the clausal form translation the clauses are reordered I times using a simple perfect shuffle algorithm.
4. scut: Backtracking is restricted for alternative start clauses.
5. cut: Backtracking is restricted for alternative reduction/extension steps.
6. comp(I): Restricted backtracking is switched off when iterative deepening exceeds the active path length I.

Note that the option conj is complete only for formulae with a provable conjecture, and scut as well as cut are complete only if used in combination with comp(I). leanCoP 2.0 uses a fixed strategy scheduling that preserves completeness. The controlled iterative deepening yields a decision procedure for ground (e.g. propositional) formulae, and refutes some (invalid) first-order formulae.

2.2 Performance

leanCoP 2.0 was tested on all 3644 problems in non-clausal form (FOF division) of version 3.3.0 of the TPTP library [24]. Problems that do not have a conjecture are negated. These are exactly those problems that are either satisfiable or unsatisfiable. Equality is dealt with by adding the equality axioms. All tests were performed on a 3 GHz Xeon system running Linux 2.6 and ECLiPSe Prolog version 5.8. The time limit was set to 600 seconds.

In Table 1 the performance of leanCoP 2.0 is compared with the performance of lean*TAP* 2.3 (the first popular lean prover) [2], leanCoP 1.0 [16], SETHEO 3.3

(likely the fastest connection tableau prover so far)[2] [7], Otter 3.3 (which commonly serves as the standard benchmark) [11] and version "Dec-2007" of Prover9 (the successor of Otter)[3] [12]. The rating and the percentage of proved problems for some rating intervals are given. FNE, FEQ and PEQ are problems without, with and containing only equality, respectively. Furthermore, the number of proved problems for each domain (see [24]) that contains at least ten problems is shown. The number of problems that are refuted, result in a time out or error, e.g. stack overflow or empty set-of-support, are listed in the last three lines.

Table 1. Performance of leanCoP 2.0 on the TPTP library

	lean*TAP*	leanCoP 1.0	SETHEO	Otter	leanCoP 2.0	Prover9
proved	375	1004	1192	1310	**1638**	1677
[%]	10%	28%	33%	36%	**45%**	46%
0s to 1s	351	787	864	987	**1124**	1281
1s to 10s	12	84	205	183	**123**	197
10s to 100s	11	74	62	106	**193**	141
100s to 600s	1	59	61	34	**198**	58
rating 0.0	194	397	435	455	**450**	464
rating >0.0	181	607	757	855	**1188**	1213
rating 1.0	0	0	2	7	**13**	8
0.00...0.24	22.8 %	56.2 %	63.9 %	72.2 %	**71.7 %**	72.8 %
0.25...0.49	5.9 %	26.0 %	34.2 %	39.7 %	**48.2 %**	69.9 %
0.50...0.74	2.2 %	7.1 %	8.5 %	3.0 %	**35.9 %**	28.2 %
0.75...1.00	0.4 %	0.0 %	1.5 %	0.7%	**11.2 %**	2.5 %
FNE	290	448	467	475	**491**	525
FEQ	85	556	725	835	**1147**	1152
PEQ	12	13	13	47	**30**	71
AGT	0	17	17	16	**29**	17
ALG	11	13	17	60	**32**	83
CSR	0	1	3	3	**2**	27
GEO	23	143	159	160	**171**	171
GRA	0	4	6	5	**6**	1
KRS	16	70	89	106	**105**	103
LCL	3	26	32	18	**24**	48
MED	0	0	1	5	**7**	1
MGT	11	31	41	54	**45**	62
NLP	4	3	7	6	**13**	13
NUM	1	35	59	31	**70**	49
PUZ	2	5	6	6	**6**	6
SET	25	170	197	229	**339**	276
SEU	5	125	124	149	**296**	259
SWC	14	14	65	84	**87**	101
SWV	55	142	154	157	**177**	183
SYN	201	199	205	210	**217**	267
refuted	0	1	27	0	33	0
time out	2979	2538	2134	700	1949	668
error	290	101	291	1634	24	1299

[2] For SETHEO the options -dr -reg -st were used, which showed the best performance.

[3] The most recent version "2008-04A" of Prover9 has a significant lower performance.

leanCoP 2.0 proves significantly more problems of the TPTP library than, e.g., Otter or SETHEO. The biggest improvement is made for problems with equality and problems that are rated as difficult, i.e. that have a high rating.

The MPTP challenge is a set of problems from the Mizar library translated into first-order logic [25]. The results of leanCoP 2.0 on the 252 "chainy" problems, in which irrelevant axioms and lemmata are *not* excluded, are shown in Table 2. leanCoP 2.0 deals with equality by adding the equality axioms. The resulting formulae contain up to 1700 axioms. Again the results are compared with lean*TAP* 2.3, Otter 3.3, SETHEO 3.3, leanCoP 1.0 and version "Dec-2007" of Prover9. All tests were performed on a 3 GHz Xeon system running Linux. leanCoP 2.0 solves significant more problem than all other listed systems.

Table 2. Performance of leanCoP 2.0 on the MPTP challenge (chainy)

	lean*TAP*	SETHEO	Otter	leanCoP 1.0	Prover9	leanCoP 2.0
proved	0	27	29	33	52	**88**
[%]	0%	11%	12%	13%	21%	**35%**
0s to 1s	0	15	17	14	33	**38**
1s to 10s	0	5	7	3	8	**21**
10s to 100s	0	6	5	11	6	**23**
100s to 300s	0	1	0	5	5	**6**
time out	252	225	150	219	101	164
error	0	0	73	0	99	0

leanCoP 2.0 participated in the FOF division of CASC-21, the CADE system competition. It solved more problems than four other (classical) provers, including Otter, and won the "Best Newcomer" award [23]. It solved four problems that the winning prover did not solve within the time limit of 360 seconds.

3 ileanCoP 1.2 for Intuitionistic Logic

We first provide details of the architecture and the source code of ileanCoP 1.2, before presenting performance results.

3.1 Architecture

ileanCoP is based on the clausal connection calculus for intuitionistic first-order logic [14]. It uses the classical search engine of leanCoP and an additional prefix unification algorithm [17] to unify the prefixes of the literals in every connection. This ensures that the characteristics of intuitionistic logic are respected and the given formula is intuitionistically valid (see also [6,27,28]).

ileanCoP 1.2 integrates all additional inference rules and search techniques of leanCoP 2.0 mentioned in Section 2.1. Like for the classical prover the input clauses are stored in Prolog's database: the fact `lit(L:Pre,C1,Grnd)` is stored for all input clauses C and literals L$\in C$, where Pre is the prefix of L, C1$=C\setminus\{$L$\}$ is a list of literals, Grnd is g iff C is ground and n otherwise.

The main part of the minimal source code is shown in Figure 2. The underlined text was added to the source code of leanCoP 2.0 in Figure 1; no other changes were done. The prefix unification algorithm (check_addco and prefix_unify) requires another 26 lines of Prolog code (see [17,13] for details). The full source code is available on the leanCoP website.

```
prove(I,S) :- ( \+member(scut,S) ->
    prove([(-(#)):(-[])],[],I,[],[Z,T],S) ;
    lit((#):_,G:C,_) -> prove(C,[(-(#)):(-[])],I,[],[Z,R],S),
    append(R,G,T) ), check_addco(T), prefix_unify(Z).
prove(I,S) :- member(comp(L),S), I=L -> prove(1,[]) ;
    (member(comp(_),S);retract(p)) -> J is I+1, prove(J,S).
prove([],_,_,_,[[],[]],_).
prove([L:U|C],P,I,Q,[Z,T],S) :- \+ (member(A,[L:U|C]), member(B,P),
    A==B), (-N=L;-L=N) -> ( member(D,Q), L:U==D, X=[], O=[] ;
    member(E:V,P), unify_with_occurs_check(E,N),
    \+ \+ prefix_unify([U=V]), X=[U=V], O=[] ;
    lit(N:V,M:F,H), \+ \+ prefix_unify([U=V]),
    (H=g -> true ; length(P,K), K<I -> true ;
    \+p -> assert(p), fail), prove(F,[L:U|P],I,Q,[W,R],S),
    X=[U=V|W], append(R,M,O) ), (member(cut,S) -> ! ; true),
    prove(C,P,I,[L:U|Q],[Y,J],S), append(X,Y,Z), append(J,O,T).
```

Fig. 2. The source code of the ileanCoP 1.2 core prover

Like the classical prover it is invoked with prove(1,S) where S is a list of settings (see Section 2.1). The predicate succeeds iff there is an intuitionistic connection proof for the clauses stored in Prolog's database.

3.2 Performance

ileanCoP 1.2 was tested on all 3644 problems in non-clausal form (FOF division) of version 3.3.0 of the TPTP library [24]. Formulae F that do not have a conjecture are translated to $F\Rightarrow\bot$. Equality is dealt with by adding the equality axioms. All tests were performed on a 3 GHz Xeon system running Linux 2.6 and ECLiPSe Prolog version 5.8. The time limit was set to 600 seconds.

In Table 3 the performance of ileanCoP 1.2 on the TPTP library is compared with all currently existing systems for intuitionistic first-order logic: JProver [21], the Prolog and C versions of ft [20], ileanTAP [13], ileanSeP[4] and ileanCoP 1.0 [14].[5] The figures for domains that contain at least 10 proved problems are shown.

[4] See http://www.leancop.de/ileansep/.

[5] The intuitionistic version of the Gandalf prover [26] is not included since it is neither complete nor sound (see the website of the ILTP library [19]).

Table 3. Performance of ileanCoP 1.2 on the TPTP library

	JProver 11-2005	ileanTAP 1.17	ft 1.23 (C)	ft 1.23 (Prolog)	ileanSeP 1.0	ileanCoP 1.0	ileanCoP 1.2
proved	186	255	262	278	303	733	**1127**
[%]	5%	7%	7%	8%	8%	20%	**31%**
0s to 1s	171	248	258	246	208	543	**750**
1s to 10s	6	3	2	27	52	72	**80**
10s to 100s	6	1	2	0	29	73	**96**
100s to 600s	3	3	0	5	14	45	**201**
rating 0.0	147	142	156	174	160	331	**397**
rating >0.0	39	113	106	104	143	402	**730**
0.00...0.24	13.1 %	15.3 %	16.1 %	17.0 %	17.1 %	40.5 %	**54.5 %**
0.25...0.49	0.4 %	4.5 %	5.0 %	5.8 %	9.1 %	20.9 %	**34.1 %**
0.50...0.74	0.0 %	1.2 %	0.2 %	0.0 %	0.0 %	4.0 %	**20.2 %**
0.75...1.00	0.0 %	0.3 %	0.2 %	0.0 %	0.0 %	0.0 %	**2.3 %**
FNE	180	175	194	209	178	312	**371**
FEQ	6	80	68	69	125	421	**756**
PEQ	5	6	4	2	1	8	**19**
AGT	0	0	2	2	5	13	**18**
ALG	4	6	2	0	0	7	**15**
GEO	1	8	7	29	36	132	**154**
KRS	33	19	26	26	18	42	**94**
LCL	0	1	1	0	0	22	**22**
MGT	7	6	7	9	0	24	**30**
NLP	7	11	7	7	3	3	**11**
NUM	0	2	1	0	1	30	**58**
SET	18	32	29	24	32	120	**222**
SEU	2	7	10	10	10	83	**200**
SWV	1	51	46	54	89	135	**162**
SYN	108	106	115	110	105	107	**121**
refuted	4	4	15	0	4	73	71
time out	2931	3344	852	2993	3161	2732	2355
error	523	41	2515	373	176	106	91

Table 4. Performance of ileanCoP 1.2 on the MPTP challenge (chainy)

	JProver 11-2005	ileanTAP 1.17	ft 1.23 (Prolog)	ft 1.23 (C)	ileanSeP 1.0	ileanCoP 1.0	ileanCoP 1.2
proved	0	0	0	1	2	19	**61**
[%]	0%	0%	0%	<1%	<1%	8%	**24%**
0s to 1s	0	0	0	1	0	4	**24**
1s to 10s	0	0	0	0	0	6	**20**
10s to 100s	0	0	0	0	2	4	**14**
100s to 300s	0	0	0	0	0	5	**3**
time out	235	252	58	9	250	233	191
error	17	0	194	242	0	0	0

The results of ileanCoP 1.2 on the 252 "chainy" problems of the MPTP challenge (see Section 2.2) are shown in Table 4.

ileanCoP 1.2 proves significantly more problems of the TPTP library and the MPTP challenge than any of the other systems. It even proves more problems of the AGT and NUM domain and of the MPTP challenge than Prover9, though intuitionistic logic is considered more difficult than classical logic and not all of those problems that are classical theorems are valid in intuitionistic logic.

4 Conclusion

We have presented leanCoP 2.0 and ileanCoP 1.2, theorem provers for classical and intuitionistic first-order logic. leanCoP 2.0 uses a few selected (well-known and new) empirical successful techniques for pruning the search space in connection calculi. By adding a prefix unification algorithm it is turned into the intuitionistic prover ileanCoP 1.2. Performance of both provers is in particular good for problems that contain equality or a large number of axioms. ileanCoP 1.2 achieves a performance that can even compete with some classical provers.

randoCoP [18] is an extension of leanCoP 2.0 and randomly reorders the axioms and literals of the given formula. It compensates for the drawback of restricted backtracking, which might narrow the search space too much. Repeatedly applying this reordering technique improves performance significantly.

Future work includes the integration of further proof search techniques and the adaption to other non-classical logics, like some first-order modal logics [6].

The complete source code of leanCoP 2.0 and ileanCoP 1.2 together with more information is available at `http://www.leancop.de`.

Acknowledgements. Thomas Raths contributed to the development of both leanCoP 2.0 and ileanCoP 1.2 by running comprehensive benchmark tests on several problem libraries. Geoff Sutcliffe kindly provided the SETHEO system.

References

1. Astrachan, O., Loveland, D.: METEORs: High Performance Theorem Provers Using Model Elimination. In: Bledsoe, W.W., Boyer, S. (eds.) Automated Reasoning: Essays in Honor of Woody Bledsoe, pp. 31–60. Kluwer, Amsterdam (1991)
2. Beckert, B., Posegga, J.: lean*TAP*: Lean Tableau-Based Theorem Proving. In: Bundy, A. (ed.) CADE 1994. LNCS, vol. 814, pp. 793–797. Springer, Heidelberg (1994)
3. Bibel, W.: Matings in Matrices. Commun. ACM 26, 844–852 (1983)
4. Bibel, W.: Automated Theorem Proving. Vieweg, Wiesbaden (1987)
5. Bibel, W., Brüning, S., Egly, U., Rath, T.: KoMeT. In: Bundy, A. (ed.) CADE 1994. LNCS, vol. 814, pp. 783–787. Springer, Heidelberg (1994)
6. Kreitz, C., Otten, J.: Connection-based Theorem Proving in Classical and Non-classical Logics. Journal of Universal Computer Science 5, 88–112 (1999)
7. Letz, R., Schumann, J., Bayerl, S., Bibel, W.: SETHEO: A High-Performance Theorem Prover. Journal of Automated Reasoning 8, 183–212 (1992)

8. Letz, R., Mayr, K., Goller, C.: Controlled Integration of the Cut Rule into Connection Tableaux Calculi. Journal of Automated Reasoning 13, 297–337 (1994)
9. Letz, R., Stenz, G.: Model Elimination and Connection Tableau Procedures. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, pp. 2015–2114. Elsevier, Amsterdam (2001)
10. Loveland, D.: Mechanical Theorem-Proving by Model Elimination. Journal of the ACM 15, 236–251 (1968)
11. McCune, W.: Otter 3.0 Reference Manual and Guide. Technical report ANL-94/6, Argonne National Laboratory (1994)
12. McCune, W.: Release of Prover9. In: Mile High Conference on Quasigroups, Loops and Nonassociative Systems, Technical report, Denver (2005)
13. Otten, J.: ileanTAP: An Intuitionistic Theorem Prover. In: Galmiche, D. (ed.) TABLEAUX 1997. LNCS, vol. 1227, pp. 307–312. Springer, Heidelberg (1997)
14. Otten, J.: Clausal Connection-Based Theorem Proving in Intuitionistic First-Order Logic. In: Beckert, B. (ed.) TABLEAUX 2005. LNCS (LNAI), vol. 3702, pp. 245–261. Springer, Heidelberg (2005)
15. Otten, J.: Restricting Backtracking in Connection Calculi. Technical report, Institut für Informatik, University of Potsdam (2008)
16. Otten, J., Bibel, W.: leanCoP: Lean Connection-based Theorem Proving. Journal of Symbolic Computation 36, 139–161 (2003)
17. Otten, J., Kreitz, C.: T-String-Unification: Unifying Prefixes in Non-classical Proof Methods. In: Miglioli, P., Moscato, U., Ornaghi, M., Mundici, D. (eds.) TABLEAUX 1996. LNCS, vol. 1071, pp. 244–260. Springer, Heidelberg (1996)
18. Raths, T., Otten, J.: randoCoP: Randomizing the Proof Search Order in the Connection Calculus. Technical report, Institut für Informatik, University of Potsdam (2008)
19. Raths, T., Otten, J., Kreitz, C.: The ILTP Problem Library for Intuitionistic Logic. Journal of Automated Reasoning 38, 261–271 (2007)
20. Sahlin, D., Franzen, T., Haridi, S.: An Intuitionistic Predicate Logic Theorem Prover. Journal of Logic and Computation 2, 619–656 (1992)
21. Schmitt, S., Lorigo, L., Kreitz, C., Nogin, A.: JProver: Integrating Connection-based Theorem Proving into Interactive Proof Assistants. In: Goré, R.P., Leitsch, A., Nipkow, T. (eds.) IJCAR 2001. LNCS (LNAI), vol. 2083, pp. 421–426. Springer, Heidelberg (2001)
22. Stickel, M.: A Prolog Technology Theorem Prover: Implementation by an Extended Prolog Compiler. Journal of Automated Reasoning 4, 353–380 (1988)
23. Sutcliffe, G.: The CADE-21 Automated Theorem Proving System Competition. AI Communications 21, 71–81 (2008)
24. Sutcliffe, G., Suttner, C.: The TPTP Problem Library. Journal of Automated Reasoning 21, 177–203 (1998)
25. Urban, J.: MPTP 0.2: Design, Implementation, and Initial Experiments. Journal of Automated Reasoning 37, 21–43 (2006)
26. 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)
27. Waaler, A.: Connections in Nonclassical Logics. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, pp. 1487–1578. Elsevier, Amsterdam (2001)
28. Wallen, L.: Automated Deduction in Nonclassical Logics. MIT Press, Cambridge (1990)

iProver – An Instantiation-Based Theorem Prover for First-Order Logic (System Description)

Konstantin Korovin[⋆]

The University of Manchester
School of Computer Science
korovin@cs.man.ac.uk

Abstract. iProver is an instantiation-based theorem prover which is based on Inst-Gen calculus, complete for first-order logic. One of the distinctive features of iProver is a modular combination of instantiation and propositional reasoning. In particular, any state-of-the art SAT solver can be integrated into our framework. iProver incorporates state-of-the-art implementation techniques such as indexing, redundancy elimination, semantic selection and saturation algorithms. Redundancy elimination implemented in iProver include: dismatching constraints, blocking non-proper instantiations and propositional-based simplifications. In addition to instantiation, iProver implements ordered resolution calculus and a combination of instantiation and ordered resolution. In this paper we discuss the design of iProver and related implementation issues.

1 Introduction

iProver is based on an instantiation framework for first-order logic Inst-Gen, developed in [3–5, 7]. We are working with clause logic and the main problem we are investigating is proving (un)satisfiability of sets of first-order clauses. The basic idea behind Inst-Gen is as follows. Given a set of first-order clauses S, we first produce a ground abstraction of S by mapping all variables into a distinguished constant, say $\perp$, obtaining a set of ground clauses $S\perp$. If $S\perp$ is unsatisfiable then S is also unsatisfiable and we are done. Otherwise, we need to refine the abstraction by adding new instances of clauses, witnessing unsatisfiability at the ground level. Instances are generated by an inference system called DSInst-Gen, which incorporates dismatching constraints (D) and semantic selection (S). We repeat this process until we obtain either (i) an unsatisfiable ground abstraction (this can be checked by any off-the-shelf SAT solver), or (ii) a saturated set of clauses, that is no non-redundant inference is applicable, and in this case completeness of the calculus [3, 7] implies that S is satisfiable. Moreover, if S is unsatisfiable and the inference process is fair, i.e., all persistent eligible inferences eventually become redundant, then completeness of the calculus guarantees that after a finite number of steps we obtain an unsatisfiable ground abstraction of S. The main ingredients which make this general scheme a basis for a useful implementation are the following.

⋆ Supported by The Royal Society.

1. The Instantiation calculus DSInst-Gen.
2. Redundancy elimination techniques.
3. Flexible saturation strategies.
4. Combination with other calculi, such as resolution.
5. State-of-the-art implementation techniques.

In the following sections we describe these components in more detail.

2 Instantiation Calculus

In order to define our main calculus DSInst-Gen we first need to define selection functions and dismatching constraints.

Semantic Selection. Selection functions allow us to restrict applicability of inferences to selected literals in clauses. Our selection functions are based on models of the propositional abstraction of the current set of clauses. In practice, such models are generated by the SAT solver, used for the ground reasoning. A *selection function* sel for a set of clauses S is a mapping from clauses in S to literals such that $\mathsf{sel}(C) \in C$ for each clause $C \in S$. We say that sel is based on a model $I_\bot$ of $S\bot$, if $I_\bot \models \mathsf{sel}(C)\bot$ for all $C \in S$. Thus, DSInst-Gen inferences are restricted to literals, whose propositional abstraction is true in a model for the propositional abstraction of the current set of clauses.

Dismatching constraints. In order to restrict instance generation further, we consider dismatching constraints. Among different types of constraints used in automated reasoning, dismatching constraints are particularly attractive. On the one hand they provide powerful restrictions for the instantiation calculus, and on the other, checking dismatching constraints can be efficiently implemented. A *simple dismatching constraint* is a formula $ds(\bar{s}, \bar{t})$, also denoted as $\bar{s} \lhd_{ds} \bar{t}$, where $\bar{s}$, $\bar{t}$ are two variable disjoint tuples of terms, with the following semantics. A solution to a constraint $ds(\bar{s}, \bar{t})$ is a substitution σ such that for every substitution γ, $\bar{s}\sigma \not\equiv \bar{t}\gamma$, where $\equiv$ is the syntactic equivalence. We will use conjunctions of simple dismatching constraints, called *dismatching constraints*, $\wedge_{i=1}^{n} ds(\bar{s}_i, \bar{t}_i)$, where every $\bar{t}_i$ is variable disjoint from all $\bar{s}_j$, and $\bar{t}_k$, for $i \neq k$. A substitution σ is a *solution* of a dismatching constraint $\wedge_{i=1}^{n} ds(\bar{s}_i, \bar{t}_i)$ if σ is a solution of each $ds(\bar{s}_i, \bar{t}_i)$, for $1 \leq i \leq n$. We will assume that for a constrained clause $C \mid [\ \wedge_{i=1}^{n} ds(\bar{s}_i, \bar{t}_i)\ ]$, the clause C is variable disjoint from all t_i, $1 \leq i \leq n$. A *constrained clause* $C \mid [\ \varphi\ ]$ is a clause C together with a dismatching constraint φ. An unconstrained clause C can be seen as a constrained clause with an empty constraint $C \mid [\ ]$. Let S be a set of constrained clauses, then $\widetilde{S}$ denotes the set of all *unconstrained clauses* obtained from S by dropping all constraints.

Proper instantiators. Another restriction on the instantiation calculus is that only proper instantiations need to be considered. A substitution θ is called a *proper instantiator* for an expression (literal, clause, etc.) if it maps a variable in this expression into a non-variable term.

DSInst-Gen Calculus. Now we are ready to formulate the DSInst-Gen calculus. Let S be a set of constrained clauses such that $\widetilde{S}\bot$ is consistent and let sel be a selection function based on a model $I_\bot$ of $\widetilde{S}\bot$. Then, the DSInst-Gen inference system is defined as follows.

DSInst-Gen

$$\frac{L \vee C \mid [\,\varphi\,] \quad \overline{L'} \vee D \mid [\,\psi\,]}{L \vee C \mid [\,\varphi \wedge \bar{x} \lhd_{ds} \bar{x}\theta\,] \quad (L \vee C)\theta}$$

where (i) $\bar{x}$ is a tuple of all variables in L, and
(ii) θ is the most general unifier of L and L', wlog. we assume that the domain of θ consist of all variables in L and L', and the range of θ is variable disjoint from the premises, and
(iii) $\mathsf{sel}(L \vee C) = L$ and $\mathsf{sel}(\overline{L'} \vee D) = \overline{L'}$, and
(iv) θ is a proper instantiator for L, and
(v) $\varphi\theta$ and $\psi\theta$ are both satisfiable dismatching constraints.

DSInst-Gen is a replacement rule, which is replacing the clause in the left premise by clauses in the conclusion. The clause in the right premise can be seen as a side condition. In [7] we have shown that DSInst-Gen calculus is sound and complete. DSInst-Gen is the main inference system implemented in iProver.

3 Redundancy Elimination

In [3] an abstract redundancy criterion is given which can be used to justify concrete redundancy elimination methods [7], implemented in iProver. In order to introduce redundancy notions we need some definitions. A *ground closure*, denoted as $C \cdot \sigma$, is a pair consisting of a clause C and a substitution σ grounding for C. Ground closures play a similar role in our instantiation framework as ground clauses in resolution. Let S be a set of clauses and C be a clause in S, then a ground closure $C \cdot \sigma$ is called a *ground instance* of S and we also say that the closure $C \cdot \sigma$ is a *representation* of the clause $C\sigma$ in S. A *closure ordering* is any ordering $\succ$ on closures that is total, well-founded and satisfies the following condition. If $C \cdot \sigma$ and $D \cdot \tau$ are such that $C\sigma = D\tau$ and $C\theta = D$ for some proper instantiator θ, then $C \cdot \sigma \succ D \cdot \tau$.

Let S be a set of clauses. A ground closure $C \cdot \sigma$ is called *redundant* in S if there exist ground closures $C_1 \cdot \sigma_1, \ldots, C_k \cdot \sigma_k$ that are ground instances of S such that, (1) $C_1 \cdot \sigma_1, \ldots, C_k \cdot \sigma_k \models C \cdot \sigma$, and (2) $C \cdot \sigma \succ C_i \cdot \sigma_i$, for each $1 \leq i \leq k$. A clause C (possibly non-ground) is called redundant in S if each ground closure $C \cdot \sigma$ is redundant in S. This abstract redundancy criterion can be used to justify many standard redundancy eliminations such as tautology elimination and strict subsumption, where the subsuming clause has strictly less literals than the subsumed.

Global Subsumption. One of the novel simplifications implemented in iProver is based on utilising propositional reasoning [7]. First, let us consider simplifications of ground clauses and then later we show how to extend this to the general case. Consider a set of clauses S. Let C be a ground clause we would like to simplify wrt. S. If we can show that a strict subclause $D \subsetneq C$ is entailed by S, then we can simplify C by D. Since our ground abstraction $S\bot$ is implied by S we can use $S\bot$ to approximate the entailment above. In particular, if we can show that $S\bot \models D$, this can be checked by the SAT solver, then C can be simplified by D. We call this simplification *global propositional subsumption* wrt. $S\bot$.

Global propositional subsumption

$$\frac{D \vee D'}{D}$$

where $S\bot \models D$ and D' is not empty.

Global propositional subsumption is a simplification rule, which allows us to remove the clause in the premise after adding the conclusion. Let us note that although the number of possible subclauses is exponential wrt. the number of literals, in a linear number of implication checks we can find a minimal wrt. inclusion subclause $D \subsetneq C$ such that $S\bot \models D$, or show that such a subclause does not exist. In [7] we have shown that global propositional subsumption generalises many known simplifications, such as strict subsumption and subsumption resolution.

Now we describe an extension of this idea to the general non-ground case (see [7] for details). First, we note that in the place of $S\bot$ we can use any ground set S_{gr}, implied by S. Let Σ_C be a signature consisting of an infinite number of constants not occurring in the signature of the initial set of clauses Σ. Let Ω be a set of injective substitutions mapping variables to constants in Σ_C. We call C' an Ω-instance of a clause C if $C' = C\gamma$ where $\gamma \in \Omega$. Let us assume that for any clause $C \in S$ there are some Ω-instances of C in S_{gr}. In [7] we have shown that if some Ω-instance of a given clause D is implied by S_{gr}, then S implies D. Now we can formulate an extension of global subsumption to the non-ground case:

Global subsumption (non-ground)

$$\frac{(D \vee D')\theta}{D}$$

where $S_{gr} \models D\gamma$ for some $\gamma \in \Omega$, and D' is not empty.
Global subsumption is one of the main simplifications implemented in iProver.

4 Saturation Algorithm: The Inst-Gen Loop

Now we are ready to put inferences and simplifications together into a saturation algorithm, called the Inst-Gen Loop, which is implemented in iProver.[1] As shown in Fig 1, the Inst-Gen Loop is a modification of the standard given clause algorithm, which accommodates propositional reasoning. Let us overview key components of the Inst-Gen Loop and how they are implemented in iProver. One of the main ideas of the given clause algorithm is to separate clauses into two sets, called Active and Passive, with the following properties. The set of Active clauses is such that all non-redundant inferences between clauses in Active are performed (upon selected literals). The set of Passive clauses are the clauses waiting to participate in inferences. Initially, the input clauses are preprocessed and groundings of the preprocessed clauses are added to the

[1] iProver is available at http://www.cs.man.ac.uk/˜korovink/iprover/

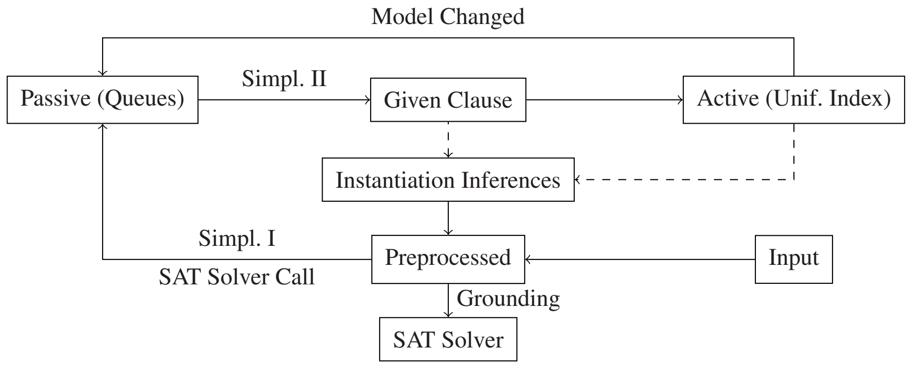


Fig. 1. The Inst-Gen Loop

SAT solver. Preprocessing currently consists of optional splitting without backtracking on variable disjoint subclauses [8]. The given clause algorithm consists of a loop and at each loop iteration the following actions are executed. First, a clause is taken from the Passive set, called the Given Clause. Then, all inferences between the Given Clause and the clauses in Active are performed and the Given Clause is moved to Active. Finally, all newly derived clauses are preprocessed and groundings of the obtained clauses are added to the SAT solver. The SAT solver is called in regular, user-defined intervals until either (i) unsatisfiability is found, in this case the input set of clauses is unsatisfiable, or (ii) all clauses are in Active, in this case the input set of clauses is satisfiable. Let us describe the components of the Inst-Gen Loop.

Passive. The Passive set are the clauses waiting to participate in inferences. Experience with resolution-based systems shows that the order in which clauses are selected for inferences from Passive is an important parameter. Usually, preference is given to the clauses which are heuristically more promising to derive the contradiction, or to the clauses on which basic operations are easier to perform. In iProver, Passive clauses are represented by two priority queues. In order to define priorities we consider numerical/boolean parameters of clauses such as: number of symbols, number of variables, age of the clause, number of literals, whether the clause is ground, conjecture distance and whether the clause contains a symbol from the conjecture (other than equality). Then, each queue is ordered by a lexicographic combination of orders defined on parameters. For example, if a user specifies an iProver option: ‘*- -inst_pass_queue1 [+age; -num_symb;+ground]*’, then priority in the first queue is given to the clauses generated at the earlier iterations of the Inst-Gen Loop (older clauses), then to the clauses with fewer number of symbols and finally to ground clauses. The clauses are taken from the queues according to a user-specified ratio.

Selection functions. Selection functions are based on the current model $I_\perp$ of the propositional abstraction of the current set of clauses. A clause can have several literals true in $I_\perp$, and the selection function can be restricted to choose one of them. Selection functions are defined by priorities based on a lexicographic combination of the literal parameters. The following parameters currently can be selected by the user: sign, ground, num_var, num_symb, and split. For example if a user specifies an iProver option: '- *-inst_lit_sel [+sign;+ground;-num_symb]*'. Then, priority (among the literals in the clause true in $I_\perp$) is given to the positive literals, then to the ground literals and then to literals with a fewer number of symbols.

Active. After the Given Clause is selected from Passive all eligible inferences between the Given Clause and clauses in Active should be performed. A unification index is used for efficient selection of clauses eligible for inferences. In particular, Active clauses are indexed by selected literals. The unification index implemented in iProver is based on non-perfect discrimination trees [6]. Let us note that since the literal selection is based on a propositional model (of a ground abstraction of the current set of clauses), selection can change during the Inst-Gen Loop iterations. This can result in moves of clauses from Active to Passive, as shown in Fig 1. Such moves can be a source of inefficiency and we minimise them by considering the selection change only in clauses participating in the current inference.

Instantiation Inferences. iProver implements DSInst-Gen calculus. In particular, constrained clauses, dismatching constraint checking and semantic-based literal selections are implemented.

Redundancy elimination. In addition to dismatching constraints, global subsumption for clauses with variables and tautology elimination are implemented.

Grounding and SAT Solver. Newly derived clauses are grounded and added to the propositional solver. Although, in our exposition we used the designated constant $\perp$ for grounding, all our arguments remain valid if we use any ground term in place of $\perp$. In particular, for grounding, iProver selects a constant with the greatest number of occurrences in the input set of clauses. After grounding, clauses are added to the propositional solver. Currently, iProver integrates MiniSat [2] solver for propositional reasoning. Incrementality of MiniSat is essential for global subsumption.

Learning Restarts. Initially, the propositional solver contains only few instances of the input clauses, and therefore selection based on the corresponding propositional model may be inadequate. Although the model and selection can be changed at the later iterations, by that time, the prover may have consumed most of the available resources. In order to overcome this, iProver implements restarts of the saturation process, keeping the generated groundings of clauses in the SAT solver. After each restart, the propositional solver will contain more instances of clauses, this can help to find a better literal selection. In addition, after each restart, global subsumption becomes more powerful.

Combination with Resolution. Instantiation, by itself, is not very well suited for generating small clauses which can be later used in simplifications such as (global) subsumption. For this, iProver implements a complete saturation algorithm for ordered resolution. The saturation algorithm for resolution is based on the same data structures as Inst-Gen Loop and implements a number of simplifications such as forward and backward subsumption (based on a vector index [11]), subsumption resolution, tautology deletion and global subsumption. Resolution is combined with instantiation by sharing the propositional solver. In particular, groundings of clauses generated by resolution are added to the propositional solver and propositional solver is used for global subsumption in both resolution and instantiation saturation loops. The user can select between combination of instantiation with resolution, pure instantiation and pure ordered resolution.

5 Implementation Details and Evaluation

iProver is implemented in a function language OCaml and integrates MiniSat solver [2] for propositional reasoning, which is implemented in C/C++. iProver v0.3.1 was evaluated on the standard benchmark for first-order theorem provers – TPTP library v3.2.0[2]. Currently, iProver does not have a built-in clausifier and we used E prover[3] for clausification. Experiments were run on a cluster of PCs with CPU 1.8GHz, Memory 512 Mb, Time Limit 300s, OS Linux v2.6.22. Out of 8984 problems in the TPTP library, iProver (single strategy) solved 4843 problems: 4000 unsatisfiable and 843 satisfiable. Problems in TPTP are rated from 0 to 1, problems with the rating 0 are easy and problems with the rating 1 cannot be solved by any state-of-the-art automated reasoning system. iProver solved 7 problems with the rating 1, and 27 with rating greater than 0.9. We compare iProver v0.2 with other systems, based on the results of the CASC-21 competition, held in 2007 [12]. In the major FOF devision, iProver is in the top three provers along with established leaders Vampire [9] and E [10]. In the effectively propositional division (EPR), iProver is on a par with the leading system Darwin [1]. We are currently working on integrating equational and theory reasoning into iProver.

References

1. Baumgartner, P., Fuchs, A., Tinelli, C.: Implementing the model evolution calculus. International Journal on Artificial Intelligence Tools 15(1), 21–52 (2006)
2. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
3. Ganzinger, H., Korovin, K.: New directions in instantiation-based theorem proving. In: Proc. 18th IEEE Symposium on LICS, pp. 55–64. IEEE Computer Society Press, Los Alamitos (2003)
4. Ganzinger, H., Korovin, K.: Integrating equational reasoning into instantiation-based theorem proving. In: Marcinkowski, J., Tarlecki, A. (eds.) CSL 2004. LNCS, vol. 3210, pp. 71–84. Springer, Heidelberg (2004)
5. Ganzinger, H., Korovin, K.: Theory Instantiation. In: Hermann, M., Voronkov, A. (eds.) LPAR 2006. LNCS (LNAI), vol. 4246, pp. 497–511. Springer, Heidelberg (2006)
6. Graf, P.: Term Indexing. LNCS, vol. 1053. Springer, Heidelberg (1996)
7. Korovin, K.: An invitation to instantiation-based reasoning: From theory to practice. In: Podelski, A., Voronkov, A., Wilhelm, R. (eds.) Volume in memoriam of Harald Ganzinger (to appear) (invited paper)
8. Riazanov, A., Voronkov, A.: Splitting without backtracking. In: Proc. of the 17 International Joint Conference on Artificial Intelligence (IJCAI 2001), pp. 611–617. Morgan Kaufmann, San Francisco (2001)
9. Riazanov, A., Voronkov, A.: The design and implementation of VAMPIRE. AI Communications 15(2-3), 91–110 (2002)
10. Schulz, S.: E - a brainiac theorem prover. AI Commun. 15(2-3), 111–126 (2002)
11. Schulz, S.: Simple and Efficient Clause Subsumption with Feature Vector Indexing. In: Sutcliffe, G., Schulz, S., Tammet, T. (eds.) Proc. of the IJCAR-2004 Workshop on Empirically Successful First-Order Theorem Proving, Cork, Ireland. ENTCS. Elsevier Science, Amsterdam (2004)
12. Sutcliffe, G.: CASC-21 proceedings of the CADE-21 ATP system competition (2007)

[2] http://www.cs.miami.edu/~tptp/
[3] http://www.eprover.org/

An Experimental Evaluation of Global Caching for $\mathcal{ALC}$ (System Description)

Rajeev Goré and Linda Postniece

Computer Sciences Laboratory
The Australian National University
Canberra ACT 0200, Australia
{Rajeev.Gore,Linda.Postniece}@anu.edu.au

Abstract. Goré and Nguyen have recently given the first optimal and sound method for global caching for the description (modal) logic $\mathcal{ALC}$, and various extensions. We report on an experimental evaluation for $\mathcal{ALC}$ plus its reflexive and reflexive-transitive extensions which compares global caching, mixed caching, unsat caching and no caching, all in a single common framework implementing a depth-first search strategy. We also evaluated a version of global caching using an unrestricted search strategy which necessarily sits outside the common framework. We conclude that global caching is an improvement over the other methods in most but not all cases.

1 Introduction

Description logics are useful in knowledge representation and reasoning in artificial intelligence and in the semantic web. The description logic $\mathcal{ALC}$ is a multimodal version of the normal modal logic K and forms the basis of many useful extensions. Many applications of automated reasoning in $\mathcal{ALC}$ can be reduced to the following EXPTIME-complete $\mathcal{ALC}$-satisfiability problem: given a finite TBox Γ of "global assumptions" and a single formula φ, decide whether there is an $\mathcal{ALC}$-model which makes φ true at one world, and which makes each member of Γ true at all worlds. We assume the reader is familiar with the terminology used in practical reasoning in (modal) description logics.

Although numerous optimisations are essential for practical (automated) reasoning in description logics, caching has remained an "external" optimisation which is often "bolted on" in an ad-hoc way. Recently, Goré and Nguyen [2] have given the first optimal (EXPTIME) and sound method for global caching for $\mathcal{ALC}$, and various extensions. We report on an experimental evaluation of various caching methods for $\mathcal{ALC}$, including two versions of global caching.

Some existing caching methods are sound only when using depth-first search (DFS), so we built a generic implementation with a fixed DFS search strategy and incorporated all methods into this framework. The global caching method of Goré and Nguyen does not rely on DFS, so we built another implementation which uses only global caching, but with an unrestricted search strategy.

While our prover is only a prototype, to retain practicality, we included the following optimisations into both basic (DFS and unrestricted) frameworks:

Negation normal form: at the start, all input formulae are put into an implication-free form such that negations appear only in front of atomic formulae;

Semantic branching for atoms: an ($\sqcup$)-rule application with principal formula $p \sqcup \psi$ creates a left denominator for p as usual, but creates a right denominator containing $\{\psi, \neg p\}$ instead of one containing just ψ;

Normalisation: there are three aspects to this optimisation:

- Modus ponens: an extra step is included at each node so that a node containing φ and $\psi \sqcup \neg\varphi$ has ψ added to it immediately;
- Subsumption: if a node contains $\{\psi, \psi \sqcup \varphi\}$, then $\psi \sqcup \varphi$ is deleted;
- Implicit "and": conjunctions are flattened recursively into their conjuncts;

Backjumping: or "use-check" whereby a clash on the left branch for $X; \varphi$ of an ($\sqcup$)-rule application on $X; \varphi \sqcup \psi$ can allow us to close the right branch $X; \psi$ without further expansion because extra stored information allows us to trace the source of the "clash" in the left branch to X, meaning that the right branch is guaranteed to close in the same way. It is well-known [4] that caching can interact with certain optimisations like backjumping.

Lazy unfolding: include certain TBox axioms in a demand driven way [5].

The source code of our prototype is available at http://users.rsise.anu.edu.au/~linda/CWB.html via the world wide web.

2 Programming Language and Basic Data Structures

Our prototype is written in C++ using the STL data structures.

Given a finite TBox Γ and an initial formula φ, the satisfiability problem has exponential complexity w.r.t. the length n of $X_0 = \Gamma \cup \{\varphi\}$. The set $\mathrm{Sf}(X_0)$ of subformulae has cardinality at most n, the number of subformulae and their negations is $2n$ and the search space contains at most 2^{2n} different nodes.

Our nodes (formula sets) consist of plain formulae and each node has a unique identifier, which is just an integer. Thus there are no world/individual "names" or "labels", meaning that we cannot handle ABoxes.

We let $\leq$ be the order in which the formulae are encountered during parsing, so "the i^{th} formula" is unique. Each tableau node carries a bit-string of length $2n$ with the i^{th} bit set to 1/0 if the i^{th} formula is present/absent from this node. The nodes are stored in a red-black tree using `std::map`, so finding a particular node requires $log_2(2^{2n}) = 2n$ (i.e. polynomial) time.

A rule is applied to a node only if applying that rule produces at least one formula new to the node. Before adding a new node to the tableau, the contents of the node are normalised using the normalisation steps outlined above.

3 Various Caching Methods

By "caching" we mean the storing of nodes and their status (`sat`/`unsat`) so that future computations of the nodes are avoided. By "global caching", we mean the method outlined in [2] in which no node is explored more than once, even if its status is "unknown". We assume familiarity with these notions but explain the various caching methods briefly:

No caching DFS (NC): there is no additional caching method, thus we need an explicit ancestor equality-blocking (loop-check) facility for termination;

Unsat caching DFS (UC): we maintain an explicit separate data-structure which stores every `unsat` node ever found. The procedure searches this cache before it creates a child node. If the required child is in the cache, then it must be `unsat` so the procedure backtracks, else the procedure creates the child and continues with the usual DFS procedure. A node is added to the `unsat`-cache once its status becomes `unsat`, thus this cache grows monotonically. There is a separate ancestor equality-blocking (loop check) to guarantee termination;

Mixed caching DFS (MC): we maintain an `unsat`-cache as described above but we also keep a local `sat`-cache. The `sat` cache grows monotonically as long as the DFS procedure remains in the same "and-structure" but it is emptied every time we move from the left branch of an ($\sqcup$)-rule to the right branch since unsoundness can result if we do not do so [1]. There is a separate ancestor equality-blocking (loop check) to guarantee termination. We have implemented the algorithm given by Donini and Massacci [1] in a naive way in that we explicitly copy the `sat`-cache when required. Thus there are possibilities for optimisation here such as lazy saving [5].

Global caching DFS with propagation (GC-DFS): we use DFS to explore the search-space but do not reclaim space upon backtracking as in all previously described methods. Instead, we keep a graph $\langle V, E\rangle$ of vertices whose status can be either `unexpanded`, `expanded`, `unsat` or `sat`, and propagate the status of `unsat`/`sat` nodes through the graph as described in detail in [2]. There is no extra ancestor equality-blocking loop check as global caching guarantees termination. Propagation is essential as it is easy to construct examples where omitting propagation can lead to unsoundness.

Unrestricted global caching and propagation (GC-NonDFS): instead of DFS, we explore the graph using a DFS-like strategy, see [2, Algorithm 2]. Our strategy differs from DFS in that at every node x where the applied rule gives k denominators $y_1, \cdots, y_k$, we create (or lookup) each successor y_i immediately, and place it in the queue for expansion, rather than creating only the first successor y_1 and exploring it as in DFS. Moreover, if at any time we encounter a cache hit to an unexpanded node z (which must already be in the queue), we bring z to the front of the queue, since we know that there are now at least two nodes that rely on the status of z, indicating that z should be processed sooner rather than later.

4 Problem Sets, Experiments and Results

We ran our experiments on a PC with 2.40GHz Intel(R) Core(TM)2 CPU 6600 and 2GB RAM. We tested our implementation on the K, KT and S4 problem sets

Table 1. Results for K Tests

K Tests: highest problem complexity solved in 1 2 4 8 16 seconds					
Problem	No Caching	Unsat Caching	Mixed Caching	Global Caching	NonDFSGC
k_branch_n	7 9 9 10 11	7 9 9 10 11	8 8 9 10 11	8 8 9 10 11	8 8 9 10 11
k_branch_p	16 19 21	16 19 21	16 19 21	17 20 21	17 20 21
k_d4_n	5 6 6 7 8	6 6 7 7 8	12 16 21	8 10 14 19 21	16 21
k_d4_p	7 7 7 8 9	6 6 7 7 8	21	21	21
k_dum_n	21	21	21	21	21
k_dum_p	21	21	21	21	21
k_grz_n	21	21	21	21	21
k_grz_p	21	21	21	21	21
k_lin_n	10 13 19 21	12 13 18 21	10 14 18 21	11 14 19 21	11 15 19 21
k_lin_p	21	21	21	21	21
k_path_n	3 3 3 3 3	4 4 5 5 6	6 8 10 13 17	8 11 13 18 21	8 11 13 17 21
k_path_p	3 3 4 4 4	5 6 6 7 8	6 8 10 13 16	9 10 13 18 21	8 11 14 18 21
k_ph_n	6 6 7 8 8	6 6 7 8 8	6 6 7 8 8	6 6 7 8 8	6 6 7 7 8
k_ph_p	5 5 5 6 6	5 5 5 6 6	5 5 5 6 6	5 5 5 6 6	5 5 5 6 6
k_poly_n	7 8 10 12 15	7 8 10 12 15	10 14 17 21	12 14 17 21	12 14 17 21
k_poly_p	12 16 19 21	13 16 19 21	13 16 19 21	11 16 18 21	11 16 18 21
k_t4p_n	4 7 9 12 16	5 7 9 13 18	10 14 20 21	12 17 21	14 18 21
k_t4p_p	6 8 12 16 20	9 16 21	9 19 21	13 20 21	13 19 21
blank count	24	26	34	33	36

from the Logics Work Bench (LWB) benchmarks [3] and the K TBox problem set from the DL98 benchmarks `http://dl.kr.org/dl98/comparison/data.html`. The LWB is a sequent-based prover so a formula φ is "provable" if $\{\neg\varphi\}$ is unsatisfiable, and φ is "not provable" otherwise: hence the appellation "p" or "n" on the problem sets. The LWB problems are actually for *monomodal* K, KT and S4 but are adequate for testing the *multimodal* logic ALC because the caching methods never inspect the value of the role name (accessibility relation). However, the LWB problems do not contain TBox axioms, therefore we also used the DL98 TBox problem set. We also tested our implementation on the galen1.tkb from the DL98 data set, which is a significantly scaled down version of the Galen ontology, containing TBox axioms and multiple roles (modalities).

Each table to follow contains a column for each of the five different caching methods, with these five methods classified into two classes. The "Unsophisticated" ones comprising of the NC and UC methods and the "Sophisticated" ones comprising of the MC and GC-DFS and GC-NonDFS methods.

For each caching method, there are five entries corresponding to time-outs of 1 second, 2 seconds, 4 seconds, 8 seconds and 16 seconds. Thus a shift of one place to the right in any single sequence of five entries corresponds to a doubling of the time-out allowed. Each such entry is an integer between 1 and 21 indicating the most complex problem that could be solved in the given time-out. Since the time-outs increase monotonically from left to right, we have replaced consecutive occurrences of "21" by blanks. Thus, within a particular caching method, the more blanks the better since this indicates that that method could

Table 2. Results for KT Tests

KT Tests: highest problem complexity solved in 1 2 4 8 16 seconds					
Problem	No Caching	Unsat Caching	Mixed Caching	Global Caching	NonDFSGC
kt_45_n	6 6 7 8 9	6 6 7 8 9	21	19 21	21
kt_45_p	4 5 5 6 6	4 5 5 6 6	21	21	21
kt_branch_n	7 8 9 10 11	7 8 9 10 11	8 8 9 10 11	7 8 9 10 11	8 8 9 10 11
kt_branch_p	21	21	21	21	21
kt_dum_n	9 11 11 13 15	9 11 11 13 14	21	21	21
kt_dum_p	21 21 21 21 21	14 15 15 17 17	21	21	21
kt_grz_n	21	21	21	21	21
kt_grz_p	21	21	21	21	21
kt_md_n	4 4 4 4 4	4 4 4 4 4	6 6 6 6 6	6 6 6 6 7	6 6 6 6 7
kt_md_p	3 3 4 4 4	3 4 4 4 4	4 4 4 4 4	4 4 4 4 4	4 4 4 4 4
kt_path_n	2 2 2 2 2	2 2 2 3 3	2 2 3 3 3	2 2 3 3 3	2 3 3 3 4
kt_path_p	3 3 3 3 3	3 3 3 3 3	3 3 3 3 3	3 3 3 3 4	3 3 4 4 4
kt_ph_n	5 5 6 7 7	5 5 6 7 7	5 5 6 7 7	5 5 6 7 7	5 5 6 7 7
kt_ph_p	5 5 5 6 6	5 5 5 6 6	5 5 5 6 6	5 5 5 6 6	5 5 5 6 6
kt_poly_n	1 2 2 2 2	1 2 2 2 2	3 3 4 4 5	4 4 4 6 6	3 4 5 6 7
kt_poly_p	7 8 10 12 15	7 8 11 13 15	7 8 11 13 15	7 8 10 13 15	7 8 10 13 15
kt_t4p_n	1 1 2 2 2	1 1 2 2 2	2 3 4 5 8	3 4 6 8 12	4 5 7 10 14
kt_t4p_p	2 2 3 4 4	3 3 4 5 5	7 9 13 19 21	9 12 18 21	7 9 14 19 21
blank count	12	12	28	28	28

solve the most complex problem using the smallest time-out. The bottom-most row of each table shows the number of "blanks" for that caching method over all problem instances: a rather crude measure of overall performance.

Finally, we note that differences of 1 integer value are not significant as we noticed such differences over two identical runs at different times, particularly for the 1 second, 2 second and 4 second time-outs.

Results for K. Given the ±1 fluctuations, Table 1 shows the following trends. Sophisticated caching is better than Unsophisticated caching. Within the Sophisticated caching methods, the best of GC-DFS and GC-NonDFS is usually better than MC. We also see that the GC-NonDFS method significantly outperforms MC for problems k_d4_n and k_t4p_n. There appears to be no clear winner between the two "global caching" methods GC-DFS and GC-NonDFS, although GC-NonDFS has the highest blank-count (due to k_d4_n).

Results for KT. The general trend continues in that the Sophisticated methods outperform the Unsophisticated methods for most problems, but for many problems, the differences are not great. The best of the two global caching methods (GC-DFS and GC-NonDFS) continues to outperform MC. Specifically, for kt_t4p_n, GC-NonDFS significantly outperforms MC, and for kt_t4p_p, GC-DFS outperforms MC.

Results for S4. As for K and KT, most of the results get better as we move from the Unsophisticated to the Sophisticated methods. However, something strange

Table 3. Results for S4 Tests

S4 Tests: highest problem complexity solved in 1 2 4 8 16 seconds					
Problem	No Caching	Unsat Caching	Mixed Caching	Global Caching	NonDFSGC
s4_45_n	12 14 18 21	11 14 18 21	13 17 21	11 14 19 21	12 15 19 21
s4_45_p	11 12 15 20 21	11 12 15 19 21	14 18 21	15 19 21	14 19 21
s4_branch_n	7 8 9 10 10	7 8 9 10 10	7 8 9 10 10	7 8 9 10 10	8 8 9 10 10
s4_branch_p	21	21	21	21	21
s4_grz_n	21	21	21	8 9 11 13 14	7 8 10 11 13
s4_grz_p	21	21	21	21	21
s4_ipc_n	4 5 5 5 5	4 5 5 5 5	7 7 7 8 8	6 6 7 8 8	6 6 7 8 8
s4_ipc_p	18 21	18 21	18 21	19 21	17 21
s4_md_n	6 6 6 6 6	6 6 6 6 7	7 8 8 9 9	7 8 8 9 9	7 8 8 9 9
s4_md_p	5 6 6 7 8	5 6 6 7 8	5 6 6 7 8	5 6 6 7 8	5 6 6 7 8
s4_path_n	1 1 1 1 1	1 1 1 1 1	2 2 2 3 3	1 2 2 2 2	1 2 2 2 2
s4_path_p	2 2 2 2 2	2 2 2 2 2	3 3 3 3 3	2 2 3 3 3	2 3 3 3 3
s4_ph_n	4 4 4 4 4	4 4 4 4 4	5 5 5 5 6	3 3 3 3 3	3 3 3 3 3
s4_ph_p	5 5 5 5 6	5 5 5 5 6	5 5 5 5 6	5 5 5 5 6	5 5 5 5 6
s4_s5_n	3 3 3 4 5	3 3 3 4 5	4 4 5 6 6	3 4 5 5 6	4 4 5 6 7
s4_s5_p	21	21	21	21	21
s4_t4p_n	1 2 2 3 4	2 3 5 6 8	4 6 9 12 15	2 4 5 7 10	2 4 5 7 9
s4_t4p_p	3 4 5 6 7	7 9 12 15 20	9 13 17 21	10 13 18 21	9 13 17 21
blank count	20	20	24	19	19

happens for s4_grz_n, where we see that global caching performs significantly worse in both of its incarnations. Another strange result is s4_t4p_n, where mixed caching significantly outperforms all others. This time, the MC method has the highest blank-count, but this is mostly due to its better performance on s4_grz_n.

Results for K TBox. The LWB benchmarks for K, KT and S4 do not contain TBox axioms, so they do not display the EXPTIME behaviour of ALC. We therefore used the TBox problem set from the DL98 benchmarks as shown in Table 4. For many problems, there is no significant difference between the Unsophisticated and Sophisticated methods. For k_d4_p, both methods of global caching significantly outperform mixed caching, and for k_lin_n, GC-DFS beats MC. For the other problems, the best of the two global caching methods is on par with mixed caching.

Galen Test. We evaluated mixed caching and global caching on the "galen1.tkb" problem from the DL'98 benchmarks which contains multiple modalities. We found that MC and both types of global caching took around 20 seconds. Moreover, MC accessed about 3 times as many nodes as GC-NonDFS.

5 Conclusions and Further Work

It is clear that global caching is a promising optimisation which deserves further investigation since it is on par or better than the other caching methods on all

Table 4. Results for K TBox Tests

K TBox Tests: highest complexity problem solved in 1 2 4 8 16 seconds																									
Problem	No Caching					Unsat Caching					Mixed Caching					Global Caching					NonDFSGC				
k_branch_n	3	3	4	5	5	3	4	4	5	5	3	4	4	5	5	3	4	4	5	5	3	4	4	5	5
k_branch_p	5	5	6	6	7	5	5	6	6	7	5	5	6	6	7	5	5	6	6	7	5	5	6	6	7
k_d4_n	2	3	3	3	4	2	3	3	3	4	3	3	3	4	4	3	3	3	4	5	2	3	3	3	4
k_d4_p	6	6	6	7	7	4	5	6	8	10	4	5	6	8	10	16	18	21			14	19	21		
k_dum_n	7	8	9	9	10	7	8	9	10	10	17	21				18	21				16	21			
k_dum_p	21					21					21					21					21				
k_grz_n	9	12	15	20	21	11	12	15	20	21	11	14	17	21		11	13	17	21		10	13	17	21	
k_grz_p	3	6	6	9	11	4	9	11	13	15	4	9	11	13	15	7	9	11	14	15	6	7	10	10	14
k_lin_n	5	7	9	12	16	5	5	6	8	10	5	5	6	8	10	5	6	8	12	16	4	4	5	5	6
k_lin_p	3	5	5	6	6	3	4	5	6	6	3	4	5	6	6	5	5	5	8	8	5	5	5	6	8
k_path_n	2	3	3	3	4	3	3	4	4	5	5	6	6	8	12	6	7	8	11	13	5	6	8	10	13
k_path_p	4	4	4	4	6	4	4	6	6	7	4	6	6	7	9	5	7	8	10	13	5	7	8	10	13
k_ph_n	5	5	6	6	7	5	5	6	6	7	5	5	6	6	7	5	5	6	6	7	5	5	6	6	7
k_ph_p	3	3	3	3	3	3	3	4	4	4	3	3	4	4	4	3	3	4	4	4	3	3	4	4	4
k_poly_n	6	8	8	11	13	6	8	9	10	13	11	15	19	21		11	14	18	21		12	14	18	21	
k_poly_p	16	19	21			13	19	21			16	19	21			16	19	21			16	19	21		
k_t4p_n	2	2	3	5	7	2	3	4	5	7	5	7	10	13	19	4	7	9	13	18	5	6	9	13	19
k_t4p_p	5	7	10	13	21	2	4	7	8	12	5	6	10	15	21	7	8	9	11	17	5	5	7	12	17
blank count	6					6					11					13					13				

our tests except s4_grz_n and s4_t4p_n. In particular, it is vital to investigate good heuristics for GC-NonDFS, since it is not tied to the DFS framework.

We observed that the best global caching method beats MC in most cases. This may be because MC spends more time per node since it must look up both the `unsat`- and `sat`-caches for every node. As noted earlier, further work is required to make the `sat`-cache of MC more efficient when descending into ($\sqcup$)-branches, by implementing lazy saving for example.

Acknowledgements. We thank Florian Widmann for many useful suggestions on numerous aspects of this work, and Linh Anh Nguyen for many clarifications.

References

1. Donini, F., Massacci, F.: EXPTIME tableaux for $\mathcal{ALC}$. AIJ 124, 87–138 (2000)
2. Goré, R., Nguyen, L.A.: Exptime tableaux for ALC using sound global caching. In: Proc. DL 2007 (June 2007), `http://dl.kr.org/dl2007/`
3. Heuerding, A., Schwendimann, S.: A benchmark method for the propositional logics K, KT, S4. Technical report, Universitaet Bern, Switzerland (1996)
4. Horrocks, I., Patel-Schneider, P.F.: Optimizing description logic subsumption. Journal of Logic and Computation 9(3), 267–293 (1999)
5. Tsarkov, D., Horrocks, I., Patel-Schneider, P.F.: Optimizing terminological reasoning for expressive description logics. JAR 39(3), 277–316 (2007)

Multi-completion with Termination Tools (System Description)*

Haruhiko Sato[1], Sarah Winkler[2], Masahito Kurihara[1], and Aart Middeldorp[2]

[1] Graduate School of Information Science and Technology
Hokkaido University, Japan
[2] Institute of Computer Science
University of Innsbruck, Austria

Abstract. In this paper we describe a new tool for performing Knuth-Bendix completion with automatic termination tools. It is based on two ingredients: (1) the inference system for completion with multiple reduction orderings introduced by Kurihara and Kondo (1999) and (2) the inference system for completion with external termination provers proposed by Wehrman, Stump and Westbrook (2006) and implemented in the Slothrop system. Our tool can be used with any termination tool that satisfies certain minimal requirements. Preliminary experimental results show the potential of our tool.

1 Introduction

Knuth and Bendix [2] introduced in a landmark paper the completion procedure which aims to transform a given set of equations into a confluent and terminating rewrite system, which can then be used to decide validity problems. The procedure takes as input a reduction order which is used to orient rules. The success of the procedure depends very much on the choice of the reduction order.

Kurihara and Kondo [3] introduced a variant that works for multiple reduction orders. It simulates the parallel execution of completion processes with the individual orders. The common features of the different processes are captured in a special data structure and corresponding inference rules. The resulting procedure is more efficient than a parallel execution of completion procedures.

Wehrman, Stump and Westbrook [7] take a different approach. Instead of relying on a reduction order supplied by the user, they call an external termination prover to orient rules. Since modern termination provers typically combine a number of different powerful techniques, this opens the way to complete systems that cannot be handled by traditional implementations of the completion procedure. One such system is CGE_2, the theory of two commuting group endomorphisms, that is completed by the Slothrop tool described in [7] without user interaction.

* This research is partially supported by FWF (Austrian Science Fund) project P18763 and JSPS Grant-in-Aid for Scientific Research (C), No. 19500020.

$$\text{orient} \quad \frac{(E \cup \{s \approx t\}, R, C)}{(E, R \cup \{s \to t\}, C \cup \{s \to t\})} \quad \text{if } C \cup \{s \to t\} \text{ terminates}$$

Fig. 1. Inference rule orient of KBtt

We combine the two approaches sketched above in a single procedure. The underlying inference system is described in the next section. In Section 3 we present implementation details and in Section 4 we present the interface of our tool. Preliminary experimental results given in Section 5 show the advantage of our approach.

2 Inference System

Figure 1 shows the orient inference rule of KBtt ($\mathcal{A}$ in [7]), the inference system underlying Slothrop. KBtt operates on triples (E, R, C) consisting of unoriented equations in E and rewrite rules in R and C.

In the orient rule, termination is checked of the combination of the new rewrite rule $s \to t$ and all previously added rewrite rules, which are stored in the third component C. (The other inference rules do not modify C.) Checking termination of the combination of $s \to t$ and the present rules in R would be unsound [4]. When both $C \cup \{s \to t\}$ and $C \cup \{t \to s\}$ can be proved terminating, in an implementation one faces the question which branch to explore. Slothrop uses a best-first search strategy in connection with a cost function that determines which branch to advance [7, Section 4].

The advantage of our approach is that both branches are explored simultaneously by incorporating the ideas of Kurihara and Kondo [3]. The completion procedure described in their paper simulates the execution of multiple completion processes in parallel. Whenever a process encounters an equation that can be oriented in either direction, the process is split into two child processes. As a process corresponds to a sequence of decisions on how to orient equations, processes will in the following be considered as bit strings. The set of all processes will be denoted by $\mathcal{P}$ and the initial process is naturally represented by the empty string ϵ.

Definition 1. *The inference system MKBtt operates on sets of* nodes*. A node $\langle s : t, R_1, R_2, E, C_1, C_2\rangle$ consists of an ordered pair of terms $s : t$ and sets of processes $R_1, R_2, E, C_1, C_2 \subseteq \mathcal{P}$ such that R_1, R_2, and E are mutually disjoint. A node $\langle s : t, R_1, R_2, E, C_1, C_2\rangle$ is identified with $\langle t : s, R_2, R_1, E, C_2, C_1\rangle$. Given a set of equations $\mathcal{F}$, the initial node set consists of all nodes $\langle s : t, \varnothing, \varnothing, \{\epsilon\}, \varnothing, \varnothing\rangle$ such that $s \approx t$ occurs in $\mathcal{F}$. The inference rules of MKBtt are displayed in Figure 2.*

The term pair $s : t$ is also referred to as the node's *datum*, the remaining components as its *labels*. Intuitively, the sets R_1 and C_1 consist of processes where

orient
$$\frac{N \cup \{\langle s : t, R_1, R_2, E, C_1, C_2\rangle\}}{\mathrm{split}_P(N) \cup \{\langle s : t, R_1 \cup R_{lr}, R_2 \cup R_{rl}, E', C_1 \cup R_{lr}, C_2 \cup R_{rl}\rangle\}}$$

with $E_{lr}, E_{rl} \subseteq E$ such that $E_{lr} \cup E_{rl} \neq \varnothing$, $P = E_{lr} \cap E_{rl}$, $E' = E \setminus (E_{lr} \cup E_{rl})$, $C[N, p] \cup \{s \to t\}$ terminates for all $p \in E_{lr}$, $C[N, p] \cup \{t \to s\}$ terminates for all $p \in E_{rl}$, $R_{lr} = (E_{lr} \setminus E_{rl}) \cup \{p0 \mid p \in P\}$, $R_{rl} = (E_{rl} \setminus E_{lr}) \cup \{p1 \mid p \in P\}$ and where $\mathrm{split}_P(N)$ replaces every $p \in P$ in any label of any node in N by $p0$ and $p1$

delete
$$\frac{N \cup \{\langle s : s, \varnothing, \varnothing, E, \varnothing, \varnothing\rangle\}}{N}$$

deduce
$$\frac{N}{N \cup \{\langle s : t, \varnothing, \varnothing, R \cap R', \varnothing, \varnothing\rangle\}}$$

if there exist nodes $\langle l : r, R, \ldots\rangle, \langle l' : r', R', \ldots\rangle \in N$ and a term u such that $s \leftarrow_{l \to r} u \to_{l' \to r'} t$ and $R \cap R' \neq \varnothing$

rewrite$_1$
$$\frac{N \cup \{\langle s : t, R_1, R_2, E, C_1, C_2\rangle\}}{\begin{array}{l} N \cup \{\langle s : t, R_1 \setminus R, R_2, E \setminus R, C_1, C_2\rangle\} \\ \quad \cup \{\langle s : u, R_1 \cap R, \varnothing, E \cap R, C_1, C_2\rangle\}\end{array}}$$

if $\langle l : r, R, \ldots\rangle \in N$, $t \to_{l \to r} u$, $t \doteq l$, and $R \cap (R_1 \cup E) \neq \varnothing$ [a]

rewrite$_2$
$$\frac{N \cup \{\langle s : t, R_1, R_2, E, C_1, C_2\rangle\}}{\begin{array}{l} N \cup \{\langle s : t, R_1 \setminus R, R_2 \setminus R, E \setminus R, C_1, C_2\rangle\} \\ \quad \cup \{\langle s : u, R_1 \cap R, \varnothing, (R_2 \cup E) \cap R, \varnothing, \varnothing\rangle\}\end{array}}$$

if $\langle l : r, R, \ldots\rangle \in N$, $t \to_{l \to r} u$, $t \rhd l$, and $R \cap (R_1 \cup R_2 \cup E) \neq \varnothing$ [b]

gc
$$\frac{N \cup \{\langle s : t, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing\rangle\}}{N}$$

subsume
$$\frac{N \cup \{\langle s : t, R_1, R_2, E, C_1, C_2\rangle\} \cup \{\langle s' : t', R_1', R_2', E', C_1', C_2'\rangle\}}{N \cup \{\langle s : t, R_1 \cup R_1', R_2 \cup R_2', E'', C_1 \cup C_1', C_2 \cup C_2'\rangle\}}$$

if $s : t$ and $s' : t'$ are variants and $E'' = (E \setminus (R_1' \cup R_2' \cup C_1' \cup C_2')) \cup (E' \setminus (R_1 \cup R_2 \cup C_1 \cup C_2))$

[a] $t \doteq l$ specifies that t and l are variants
[b] $\rhd$ denotes the encompassment relation

Fig. 2. Inference rules of MKBtt

$s \to t$ is contained in the current set of rules and the current set of constraints for this process, respectively. The sets R_2 and C_2 play the analogous role with respect to the rule $t \to s$. The set E contains processes where $s \approx t$ is in the current set of equations. In the following we make some clarifying remarks on some of the inference rules.

– Unlike its KB counterpart, the orient rule does not orient an equation in just one direction. Instead, for the chosen node $n = \langle s : t, R_1, R_2, E, C_1, C_2\rangle$ one checks for every process p occurring in the node's equation label E if the constraint system for p remains terminating if either $s \to t$ or $t \to s$ is added. Formally

$$C[N, p] = \bigcup_{n \in N} C[n, p] \quad \text{with} \quad C[n, p] = \begin{cases} \{s' \to t'\} & \text{if } p \in C_1' \\ \{t' \to s'\} & \text{if } p \in C_2' \\ \varnothing & \text{otherwise} \end{cases}$$

  denotes the set of constraints for process p, where for every $p \in E$ it is determined whether $C[N, p]$ terminates in combination with $s \to t$ or $t \to s$. If only one orientation is possible, p is moved to the respective rule label in the node. If both orientations are possible, we have to keep track of both alternatives. Thus the process splits into two child processes $p0$ and $p1$. Each of them is put into the corresponding rule label. Moreover, $\text{split}_P(N)$ replaces p by its derivatives $p0$ and $p1$ in all non-selected nodes.
– If $l \to r$ and $l' \to r'$ allow for a peak $s \leftarrow_{l \to r} u \to_{l' \to r'} t$, deduce adds the equation $s \approx t$ to those processes that have both rules present in their rule set. In our implementation, only critical pairs are considered.
– Given a term pair $s : t$ and a rewrite step $t \to_R u$, the inference rules compose, simplify, and collapse of KBtt create an equation or rule with terms s and u. The effect of these rules is simulated by rewrite_1 and rewrite_2, as explained in [3].

It is not difficult to state and prove (partial) correctness and completeness criteria for MKBtt by relating MKBtt to KBtt, akin to [3, Section 2.2].

3 Implementation

In our implementation of MKBtt the inference rules of Figure 2 are employed according to the strategy described in Figure 3, closely resembling the algorithm proposed in [3]. The mkbTT procedure maintains two node sets, *No* containing *open* and *Nc* containing *closed* nodes. The union $No \cup Nc$ represents all nodes present in the completion process. Intuitively, every node in *Nc* has been completely exploited with respect to the inference rules orient, delete, and gc. Moreover, every pair of nodes in *Nc* has been fully exploited with respect to the inference rules that involve two nodes: deduce, $\text{rewrite}_{1,2}$ and subsume. Initially, *No* contains all nodes and *Nc* is empty. Below we shortly comment on the functions involved in mkbTT and their implementation.

– At the start of every recursive call of mkbTT, it is checked whether some process p was successful. This is the case if every equation was oriented and every rule was fully considered with respect to the inferences involving two nodes, i.e., if all of $E[Nc, p]$, $R[No, p]$, and $E[No, p]$ are empty.
– If no successful process was found, *choose* selects an open node. The measure applied in this selection has considerable impact on the overall performance.

```
procedure $mkbTT(No,\ Nc)$
if $success$ then                return successful $p$
else if $No = \varnothing$ then  $fail$
else $n := choose(No)$;
    $No = add(delete(rewrite(\{n\}, Nc)), No \setminus \{n\})$;
    if $n \neq \langle \ldots, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing \rangle$ then
      $(n, No, Nc) := orient(n, No, Nc)$;
      if $n \neq \langle \ldots, \varnothing, \varnothing, \ldots, \ldots, \ldots \rangle$ then
        $No := add(delete(rewrite(Nc, \{n\})), No)$;   $Nc := gc(Nc)$;
        $No := add(delete(deduce(n, Nc)), No)$;
        $No := gc(add(rewrite(No, Nc), No))$;          $Nc := add(\{n\}, Nc)$;
    $mkbTT(No, Nc)$;
```

Fig. 3. Procedure implementing MKBtt

In our implementation we first choose a process p for which $\mid E[Nc \cup No, p] \mid + \mid R[Nc \cup No, p] \mid$ is minimal and then a node for this process by considering the term size and timestamp, the latter to ensure fairness of the derivation.

- $rewrite(N, N')$ applies $\mathsf{rewrite}_{1,2}$ to nodes in N by using rules in N'. Nodes are considered as mutable structures. Thus the node objects in N are modified and only newly created nodes are returned.
- Immediately after rewriting, *delete* is called, applying the corresponding inference rule to avoid creating nodes with equal terms.
- $orient(n, No, Nc)$ applies the inference rule orient to n and, if required, splits processes occurring in labels of No and Nc. The modified node n and the node sets No and Nc are returned.
- $gc(N)$ removes nodes from N where all labels are empty.
- $deduce(n, N)$ returns nodes derived from the respective inference such that at least one rule comes from n.
- $add(N, N')$ merges the nodes in N into N' such that $\mathsf{subsume}$ is fully exploited. Only inferences where at least one node comes from N have to be considered.

4 Interface

The mkbTT procedure is implemented in OCaml. A binary compiled for Linux is available from `http://cl-informatik.uibk.ac.at/mkbtt`. The tool is equipped with a simple command-line interface. The termination prover is given as argument to the `-tp` option. Any termination prover that adheres to the format of the International Competitions of Termination Tools[1] can be used: an executable that takes as argument the name of a file describing the termination problem in the TPDB[2] format and prints `YES` on the first line of the output if termination could

[1] `http://www.lri.fr/~marche/termination-competition/`
[2] Termination Problem Data Base, `http://www.lri.fr/~marche/tpdb/`

```
SUCCESS
246.64 (total time)

STATISTICS
number of inference steps: 77
orient:      218.09         rewrite:      19.25
deduce:        2.89         termination: 209.64

external termination prover: ttt2fast
calls to termination prover: 1072 (yes: 933, timeouts: 0)
time limit per call:         1.0
```

Fig. 4. Sample output (slightly reformatted)

be established. Our tool accepts two time limits: for the overall procedure (specified with `-t`) and for each call to the termination prover (`-T`). Further options are `-ct` to print the completed system and `-st` to obtain some useful statistics. Figure 4 shows the output for the call

```
mkbtt -t 3600 -T 1 -st -tp ttt2fast WSW06_CGE2.trs
```

5 Experimental Results

In Table 1 we present some experimental data.[3] All tests were performed on a workstation equipped with an Intel® Pentium™ M processor running at a CPU rate of 2 GHz on 1 GB of system memory and with a time limit of 1 hour. Column (1) shows the total time in seconds, column (2) the percentage spent on termination, column (3) the number of calls to the external termination prover, and column (4) the number of inference steps. A timeout is indicated by ∞.

In the first Slothrop and MKBtt blocks, AProVE [1] is used as termination prover with a time limit of 5 seconds per call. (We modified the function for termination checks in the Slothrop source[4] such that the same back-end can be used with both approaches.) In the second Slothrop and MKBtt blocks we use a special version of $\mathsf{T_TT_2}$[5] with a time limit of 1 second per call. This version of $\mathsf{T_TT_2}$, which uses dependency pairs and the recursive SCC algorithm together with the subterm criterion and some simple strategies like counting function symbols and linear polynomials, is considerably weaker than AProVE when it comes to termination proving power, but also considerably faster. We anticipate that further savings can be achieved by more tightly coupling the termination prover and the MKBtt code.

As can be seen from line SL-cge2 in Table 1, completing the theory of two communicating group endomorphism (CGE$_2$), which is termed Slothrop*'s defining achievement* in [6], takes about 246 seconds using MKBtt with the fast variant

[3] Further details can be obtained from `http://cl-informatik.uibk.ac.at/mkbtt`.

[4] `http://www.cs.utexas.edu/~iwehrman/slothrop-1.1.0-src.tar.gz`

[5] `http://colo6-c703.uibk.ac.at/ttt2`

Table 1. Experimental Results

	SLOTHROP			MKBTT				SLOTHROP			MKBTT			
TRS	(1)	(2)	(3)	(1)	(2)	(3)	(4)	(1)	(2)	(3)	(1)	(2)	(3)	(4)
SK90_3.01	800.37	97	326	85.30	99	51	29	71.52	67	304	4.45	79	51	29
SK90_3.03	163.51	98	86	90.97	98	53	29	9.97	65	86	6.19	63	53	29
SK90_3.04	∞			∞				∞			508.24	55	658	140
SK90_3.05	∞			913.77	98	225	94	78.48	32	258	46.45	43	220	92
SK90_3.06	∞			∞				∞			44.22	74	290	75
SK90_3.07	∞			513.27	99	242	67	∞			46.03	68	282	77
SK90_3.19	84.25	99	43	112.65	99	65	11	3.39	90	43	5.01	96	65	11
SK90_3.22	∞			∞				∞			617.33	33	1406	141
SK90_3.27	∞			∞				73.61	95	70	118.56	65	143	37
SL-ack	12.08	99	8	13.26	99	10	5	0.24	96	8	0.33	91	10	5
SL-cge2	∞			2793.21	99	1220	77	665.29	36	1384	246.64	85	1072	77
SL-cge3	∞			∞				∞			∞			
SL-endo	246.56	95	101	218.96	99	135	45	12.64	39	105	7.87	74	135	45
SL-ep	∞			∞				54.47	82	266	230.06	92	1101	26
SL-groups	46.71	97	30	96.94	99	49	35	2.10	46	30	2.28	71	49	35

of $\mathsf{T_TT_2}$. We also managed to complete CGE$_3$, which has not been achieved with SLOTHROP, although it takes nearly 2 hours. (The completed system is slightly different from the one described in [5], which was obtained by hand.)

References

1. 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 (LNAI), vol. 4130, pp. 281–286. Springer, Heidelberg (2006)
2. Knuth, D.E., Bendix, P.: Simple word problems in universal algebras. In: Leech, J. (ed.) Computational Problems in Abstract Algebra, pp. 263–297. Pergamon Press, Oxford (1970)
3. Kurihara, M., Kondo, H.: Completion for multiple reduction orderings. Journal of Automated Reasoning 23(1), 25–42 (1999)
4. Sattler-Klein, A.: About changing the ordering during Knuth-Bendix completion. In: Enjalbert, P., Mayr, E.W., Wagner, K.W. (eds.) STACS 1994. LNCS, vol. 775, pp. 175–186. Springer, Heidelberg (1994)
5. Stump, A., Löchner, B.: Knuth-Bendix completion of theories of commuting group endomorphisms. Information Processing Letters 98(5), 195–198 (2006)
6. Wehrman, I.: Knuth-Bendix completion with modern termination checking. Master's thesis, Washington University in St. Louis, Technical report WUCSE-2006-45 (2006)
7. Wehrman, I., Stump, A., Westbrook, E.M.: Slothrop: Knuth-Bendix completion with a modern termination checker. In: Pfenning, F. (ed.) RTA 2006. LNCS, vol. 4098, pp. 287–296. Springer, Heidelberg (2006)

MTT: The Maude Termination Tool (System Description)*

Francisco Durán[1], Salvador Lucas[2], and José Meseguer[3]

[1] LCC, Universidad de Málaga, Spain
[2] DSIC, Universidad Politécnica de Valencia, Spain
[3] CS Dept., University of Illinois at Urbana-Champaign, USA

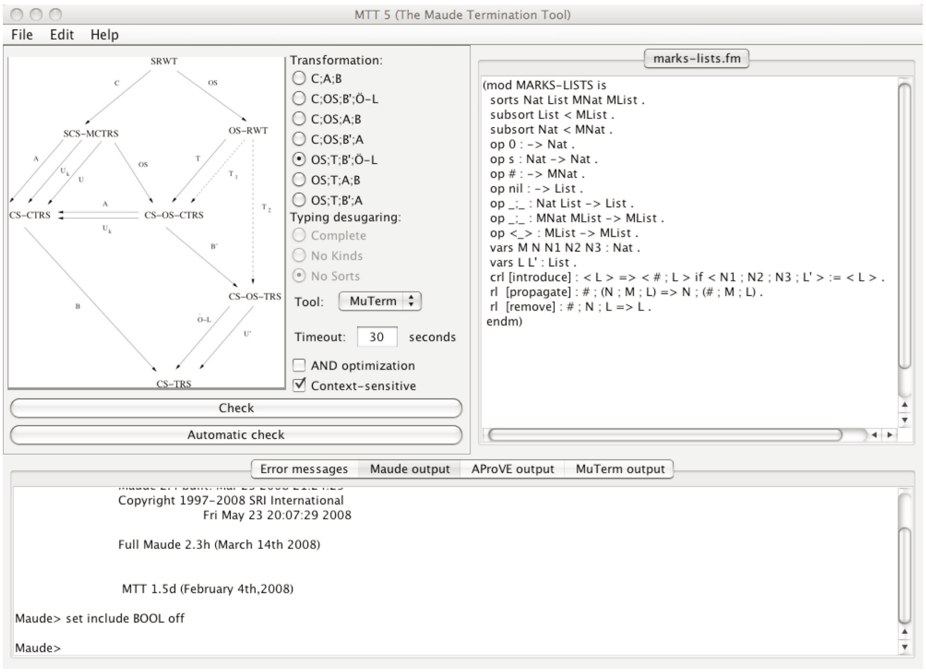
1 Introduction

Despite the remarkable development of the theory of termination of rewriting, its application to high-level programming languages is far from being optimal. This is due to the need for features such as conditional equations and rules, types and subtypes, (possibly programmable) strategies for controlling the execution, matching modulo axioms, and so on, that are used in many programs and tend to place such programs outside the scope of current termination tools. The operational meaning of such features is often formalized in a proof-theoretic manner by means of an inference system (see, e.g., [2, 3, 17]) rather than just by a rewriting relation. In particular, Generalized Rewrite Theories (GRT) [3] are a recent generalization of rewrite theories at the heart of the most recent formulation of Maude [4]. The corresponding termination notions can also differ from the standard ones, see [14]. For these reasons, although there is a good number of tools which can be used to automatically prove termination of rewriting (e.g., [9], [13], ...) they cannot directly prove termination of, e.g., Elan [1], Maude [4] or CafeOBJ [8] programs. As an illustrative example, consider the Maude module `MARKS-LISTS` in the MTT snapshot in Figure 1. Given a list representation of a multiset of natural numbers it (nondeterministically) computes its submultisets of size 2. A mark '`#`' is *introduced* into a given `List` of numbers (of sort `Nat`) to yield a *marked* list of sort `MList` (supersort of `List`). The matching condition `< N1 ; N2 ; N3 ; L' > := < L >` in the conditional rule ensures that '`#`' is introduced into lists of at least three elements. Symbol `#` is intended to mark a number to be *removed* by using the third rule (thus producing a *sublist* of the original one). The mark is *propagated* inside the structure of the list until it is finally *removed* (together with its companion number) to produce a list of sort `List` on which we can restart the process. Objects from both `List` and `MList` can be built by using a single *overloaded* constructor `_;_`.

Modeling `MARKS-LISTS` as a Conditional Term Rewriting System (CTRS, see [18]) by translating the matching condition into a rewriting condition as:

$$\texttt{< L >} \to \texttt{< \# ; L >} \text{ if } \texttt{< L >} \to \texttt{< N1 ; N2 ; N3 ; L' >}$$

* Work partially supported by the EU (FEDER) and Spanish MEC under grants TIN 2005-09405-C02-01 and TIN 2007-68093-C02-02; José Meseguer was partially supported by ONR grant N00014-02-1-0715 and NSF Grant CCR-0234524.



Transformation: C;A;B / C;OS;B';Ö-L / C;OS;A;B / C;OS;B';A / [✓ OS;T;B';Ö-L] / OS;T;A;B / OS;T;B';A

Typing desugaring: Complete / No Kinds / [✓ No Sorts]

Tool: MuTerm

Timeout: 30 seconds

AND optimization

[✓ Context-sensitive]

```
(mod MARKS-LISTS is
  sorts Nat List MNat MList .
  subsort List < MList .
  subsort Nat < MNat .
  op 0 : -> Nat .
  op s : Nat -> Nat .
  op # : -> MNat .
  op nil : -> List .
  op _;_ : Nat List -> List .
  op _;_ : MNat MList -> MList .
  op <_> : MList -> MList .
  vars M N N1 N2 N3 : Nat .
  vars L L' : List .
  crl [introduce] : < L > => < # ; L > if < N1 ; N2 ; N3 ; L' > := < L > .
  rl  [propagate] : # ; (N ; M ; L) => N ; (# ; M ; L) .
  rl  [remove] : # ; N ; L => L .
 endm)
```

```
                Copyright 1997-2008 SRI International
                           Fri May 23 20:07:29 2008

                Full Maude 2.3h (March 14th 2008)


                 MTT 1.5d (February 4th,2008)

Maude> set include BOOL off

Maude>
```

Fig. 1. MTT snapshot with module `MARKS-LISTS`

yields a *nonterminating* system. The application of this rule requires the reduction of (an instance of) `< L >` into (an instance of) `< N1 ; N2 ; N3 ; L' >` to satisfy the condition. Since the left-hand side `< L >` of the conditional rule itself can also be considered in any attempt to satisfy the conditional part of the rule, we run into a nonterminating computation, see [14] for a deeper discussion of this issue.

However, viewed as a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ and executed as a Maude program, `MARKS-LISTS` is terminating! The key points here are the sort information and that solving the matching condition involves *no rewriting step*. Matching conditions are evaluated in Maude with respect to the set E of *equations* which is different from the rules R in $\mathcal{R}$. A *matching-modulo-E* semantics is given for solving matching conditions. In our `MARKS-LISTS` example, E is empty and the matching condition becomes *syntactic pattern matching*. *No reduction* is allowed! Indeed, only when the *two* kinds of E- and R-computations which are implicit in the specification are (separately!) taken into account, are we able to prove this program terminating. Other features like sort information (including both the existence of a sort hierarchy and also the association of sorts to function symbols in module `MARKS-LISTS`), memberships [2, 17], context-sensitivitiy [11, 12], rewriting modulo A-/AC-axioms, etc., can play a crucial role in the termination behavior and hence in any attempt to provide an automatic proof of it, see, e.g., [5, 6, 15].

This paper emphasizes the techniques needed to go from standard termination methods and tools to termination tools for programs in rule-based languages with expressive features. Specifically, we focus on the implementation of MTT, a new tool for proving termination of Maude programs like `MARKS-LISTS`; see:

`http://www.lcc.uma.es/~duran/MTT`

Due to lack of space, we can only give a high-level overview of the techniques involved. Detailed accounts of all the transformations used can be found in [5, 6, 7, 14, 15].

2 Proving Termination by Transformation

In [5, 6, 15], different theory transformations associating a CS-TRS, i.e., a TRS (Term Rewriting System) together with a *replacement map* μ [11, 12], to a *membership rewrite theory* [2, 17] were presented. In [7] we have generalized this methodology to rewriting logic theories $\mathcal{R} = (\Sigma, E, R)$, as the `MARKS-LISTS` one[3]. As discussed above, from the point of view of termination, the main problem with such theories is that there are two different rewrite relations (with E and with R), which cannot always be just joined, because equational conditions (for instance, matching conditions) are solved using just E.

$$\begin{tikzcd}
 & & \text{SRWT} \arrow[dl, "{[7]}"'] \arrow[dr, "{[7]}"] & & \\
 & \text{SCS–MCTRS} \arrow[dl, "{[6]}"'] \arrow[dr, "{[15]}"] & & \text{OS–RWT} \arrow[dl, "{[7]}"'] \arrow[dd, dashed, "{[7]}"] & \\
\text{CS–CTRS} \arrow[ddrr, "{[6]}"] & & \text{CS–OS–CTRS} \arrow[ll, "{[6]}"'] \arrow[dr, "{[15]}"] & & \\
 & & & \text{CS–OS–TRS} \arrow[dl, "{[15]}"'] & \\
 & & \text{CS–TRS} & &
\end{tikzcd}$$

In MTT we take advantage of all previous theoretical developments and use *sequences of transformations* which are applied in a kind of pipeline to finally obtain a CS-TRS whose termination can be proved by using existing tools such as AProVE [9] or MU-TERM [13]. In this section, we briefly discuss such theory transformations whose hierarchy is depicted in the above diagram. All the time non-termination is preserved under the transformations in such a way that a proof of termination of a system which is downwards the diagram implies termination of the system which originated it upwards. Each arc is labelled with the reference where the corresponding transformation is described. Before describing these transformations, we recall the notion of rewrite theory.

2.1 Rewrite Theories

A rewriting logic specification is called a *rewrite theory* (RWT) [3]. It is a tuple $\mathcal{R} = (\Sigma, E \cup Ax, \mu, R, \phi)$, where:

- $(\Sigma, E \cup Ax)$ is a membership equational (MEL) theory: Σ is an order-sorted signature [10], Ax is a set of (equational) axioms, and E is a set of sentences

$$t = t' \quad \text{if} \quad A_1, \ldots, A_n \qquad\qquad \text{or} \qquad\qquad t : s \quad \text{if} \quad A_1, \ldots, A_n$$

  where the A_i are atomic equations or memberships $t : s$ establishing that term t has sort s [2, 17].
- μ is a mapping specifying for each $f \in \Sigma$ the argument positions under which subterms can be simplified with the equations in E [11, 12].
- R is a set of *labeled conditional rewrite rules* of the general form

$$r : (\forall X)\ q \longrightarrow q' \ \text{ if } \ (\bigwedge_i u_i = u'_i) \wedge (\bigwedge_j v_j : s_j) \wedge (\bigwedge_l w_l \longrightarrow w'_l).$$

- $\phi : \Sigma \longrightarrow \mathbb{N}$ is a mapping assigning to each function symbol $f \in \Sigma$ (with, say, n arguments) a set $\phi(f) \subseteq \{1, \ldots, n\}$ of *frozen positions* under which it is forbidden to perform any rewrites with rules in R.

Intuitively, $\mathcal{R}$ specifies a *concurrent system*, whose states are elements of the initial algebra $T_{\Sigma/E \cup Ax}$ and whose *concurrent transitions* are specified by the rules R, subject to the frozenness constraints imposed by ϕ. Therefore, mathematically each state is modeled as an $E \cup Ax$-equivalence class $[t]_{E \cup Ax}$ of ground terms, and rewriting happens *modulo* $E \cup Ax$, that is, R rewrites not just terms t but rather $E \cup Ax$-equivalence classes $[t]_{E \cup Ax}$ representing states.

2.2 Proving Termination of Rewrite Theories with MTT

According to the diagram above, the most general kind of systems we can deal with are (sugared) rewrite theories (SRWTs). This means that (following the usual practice), we consider rewrite theories as presented by keeping the intuitive order-sorted subtyping features intact as helpful *syntactic sugar*, and adding membership axioms only when they are strictly needed (see [15, 7] for a more detailed discussion). This *order-sorted way* allows us to take advantage of existing techniques for proving termination of order-sorted context-sensitive term rewriting systems (OS-CS-TRSs) [15]. On the order hand, we can still make use of the (older) *MEL-way* where we start from a (context-sensitive) membership equational specification and then apply the methods developed in [5, 6] to obtain a proof of termination. Thus, given an SRWT, we can either [7]:

1. translate it into an order-sorted rewrite theory (OS-RWT) where memberships have been handled by using membership predicates which express different membership conditions defined by (possibly conditional) rules, or
2. translate it into a sugared CS-CTRS with conditional rewrite and membership rules, i.e., an SCS-MCTRS.

Then, we can further transform a SCS-MCTRS into either a conditional context-sensitive TRS (CS-CTRS) or a conditional order-sorted CS-TRS (OS-CS-CTRS). In the first case, the transformation is accomplished by introducing membership predicate symbols (and rules) [6]. In the second case, the transformation tries to keep the sort structure of the SCS-MCTRS untouched and membership predicate symbols are introduced only if necessary [15]. After treating membership information, the next step is dealing with conditional rules. The classical transformation from CTRSs into TRSs (see, e.g., [18]) has been generalized to deal with context-sensitivity (so that a CS-CTRS is transformed into a CS-TRS) [6] and with sorts (then, an OS-CS-CTRS is transformed into an OS-CS-TRS) [15]. Finally, we can transform an OS-CS-TRS into a CS-TRS by using the transformation discussed in [15], which is an straightforward adaptation of Ölveczky and Lysne's transformation [19]. Nowadays, proofs of termination of CS-TRSs can be achieved by using AProVE [9] or MU-TERM [13]. However, since termination of a TRS $\mathcal{R}$ implies that of the CS-TRS $(\mathcal{R}, \mu)$ for any replacement map μ, other tools for proving termination of rewriting could be used as well.

3 Implementation of the Tool

Our current tool MTT 1.5[1] takes Maude programs as inputs and tries to prove them terminating by using existing termination tools as back-ends.

MTT 1.5 can use as back-end tool any termination tool supporting the TPDB syntax and following the rules for the Termination Competition [16].[2] This allows us to interact with the different tools in a uniform way, and not restricting ourselves to a specific set of tools. Thus, tools that have participated in the competition, like AProVE, MU-TERM, TTT, etc., or that accommodate to the syntax and form of interaction, can be used as back-ends of MTT. In the MTT environment, Maude specifications can be proved terminating by using (any of these) distinct formal tools, allowing the user to choose the most appropriate one

[1] Previous versions of MTT were able to handle only membership equational programs in the Maude syntax. Only two of the currently supported transformations were available, and it was able to interact, in an *ad hoc* way, only with AProVE, MU-TERM, and CiME.

[2] The Termination Competition rules and the TPDB syntax can be found at `http://www.lri.fr/~marche/termination-competition/`

for each particular case, a combination of them, when a particular tool cannot find a proof.

Two main components can be distinguished: (1) a Maude specification that implements the theory transformations described in the diagram above, and (2) a Java application that connects Maude, and the back-end tools, and provides a graphical user interface. The Java application is in charge of sending the Maude specification introduced by the user to Maude to perform transformations; depending on the selections, one transformation or another will be accomplished. The resulting TRS may be proved terminating by using any of the available back-end tools. Notice that such resulting TRS may have associative or associative-commutative operators, context sensitive information, etc., which are expected to be appropriately handled by the selected back-end tool.

Alternatively, the MTT *expert* can be used to try an appropriate sequence of them automatically. The current expert just tries different paths in the transformations graphs sequentially.

The interaction between MTT and the back-end tools can be done in two ways:

– If the external tool is installed in the same machine, they can interact via pipes. This is the more efficient form of interaction available.
– Interaction based on web services is also possible. This is the most flexible of the possibilities offered by MTT (no local configuration is required).

MTT has a graphical user interface where one can introduce the specifications to be checked. MTT gives the output of the back-end tool used in the check. All intermediate transformed specifications can also be obtained. Some benchmarks are here: `http://www.lcc.uma.es/~duran/MTT/mtt15/Samples`.

Acknowledgements. Special thanks are due to Claude Marché and Xavier Urbain for their collaboration in the development of the first versions of MTT.

References

[1] Borovanský, P., Kirchner, C., Kirchner, H., Moreau, P.-E.: ELAN from a rewriting logic point of view. Theoretical Computer Science 285, 155–185 (2002)
[2] Bouhoula, A., Jouannaud, J.-P., Meseguer, J.: Specification and proof in membership equational logic. Theoretical Computer Science 236, 35–132 (2000)
[3] Bruni, R., Meseguer, J.: Semantic foundations for generalized rewrite theories. Theoretical Computer Science 351(1), 386–414 (2006)
[4] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
[5] Durán, F., Lucas, S., Marché, C., Meseguer, J., Urbain, X.: Proving Termination of Membership Equational Programs. In: Sestoft, P., Heintze, N. (eds.) Proc. of ACM SIGPLAN 2004 Symposium on Partial Evaluation and Program Manipulation, PEPM 2004, pp. 147–158. ACM Press, New York (2004)

[6] Durán, F., Lucas, S., Marché, C., Meseguer, J., Urbain, X.: Proving Operational Termination of Membership Equational Programs. Higher-Order and Symbolic Computation (published online) (to appear, April 2008)
[7] Durán, F., Lucas, S., Meseguer, J.: Operational Termination in Rewriting Logic. Technical Report 2008 (2008), http://www.dsic.upv.es/~slucas/tr08.pdf
[8] Futatsugi, K., Diaconescu, R.: CafeOBJ Report. AMAST Series. World Scientific (1998)
[9] Giesl, J., Schneider-Kamp, P., Thiemann, R.: AProVE1.2. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 281–286. Springer, Heidelberg (2006), http://www-i2.informatik.rwth-aachen.de/AProVE
[10] Goguen, J., Meseguer, J.: Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. Theoretical Computer Science 105, 217–273 (1992)
[11] Lucas, S.: Context-sensitive computations in functional and functional logic programs. Journal of Functional and Logic Programming 1998(1), 1–61 (1998)
[12] Lucas, S.: Context-sensitive rewriting strategies. Information and Computation 178(1), 294–343 (2002)
[13] Lucas, S.: MU-TERM: A Tool for Proving Termination of Context-Sensitive Rewriting. In: van Oostrom, V. (ed.) RTA 2004. LNCS, vol. 3091, pp. 200–209. Springer, Heidelberg (2004), http://www.dsic.upv.es/~slucas/csr/termination/muterm
[14] Lucas, S., Marché, C., Meseguer, J.: Operational termination of conditional term rewriting systems. Information Processing Letters 95(4), 446–453 (2005)
[15] Lucas, S., Meseguer, J.: Operational Termination of Membership Equational Programs: the Order-Sorted Way. In: Rosu, G. (ed.) Proc. of the 7th International Workshop on Rewriting Logic and its Applications, WRLA 2008. Electronic Notes in Theoretical Computer Science (to appear, 2008)
[16] Marché, C., Zantema, H.: The termination competition. In: Baader, F. (ed.) RTA 2007. LNCS, vol. 4533, pp. 303–313. Springer, Heidelberg (2007)
[17] Meseguer, J.: Membership algebra as a logical framework for equational specification. In: Parisi-Presicce, F. (ed.) WADT 1997. LNCS, vol. 1376, pp. 18–61. Springer, Heidelberg (1998)
[18] Ohlebusch, E.: Advanced Topics in Term Rewriting. Springer, Heidelberg (2002)
[19] Ölveczky, P.C., Lysne, O.: Order-Sorted Termination: The Unsorted Way. In: Hanus, M., Rodríguez-Artalejo, M. (eds.) ALP 1996. LNCS, vol. 1139, pp. 92–106. Springer, Heidelberg (1996)

Celf – A Logical Framework for Deductive and Concurrent Systems (System Description)

Anders Schack-Nielsen and Carsten Schürmann*

IT University of Copenhagen, Denmark

Abstract. CLF (Concurrent LF) [CPWW02a] is a logical framework for specifying and implementing deductive and concurrent systems from areas, such as programming language theory, security protocol analysis, process algebras, and logics. Celf is an implementation of the CLF type theory that extends the LF type theory by linear types to support representation of state and a monad to support representation of concurrency. It relies on the judgments-as-types methodology for specification and the interpretation of CLF signatures as concurrent logic programs [LPPW05] for experimentation. Celf is written in Standard ML and compiles with MLton, MLKit, and SML/NJ. The source code and a collection of examples are available from `http://www.twelf.org/~celf`.

1 Introduction

The Celf system is a tool for experimenting with deductive and concurrent systems prevalent in programming language theory, security protocol analysis, process algebras, and logics. It supports the specification of object language syntax and semantics through a combination of deductive methods and resource-aware concurrent multiset transition systems. Furthermore it supports the experimentation with those specifications through concurrent logic programming based on multiset rewriting with constraints.

Many case studies have been conducted in Celf including all of the motivating examples that were described in the original CLF technical report [CPWW02b]. In particular, Celf has been successfully employed as a tool for experimenting with concurrent ML, in particular its type system and a destination passing style operational semantics. Our Celf encoding provides Haskell-style suspensions with memoizations, futures, mutable references, and concurrency omitting negative acknowledgments. Furthermore, we used Celf to experiment with the design of security protocols, especially the widely studied and well-understood Needham-Schroeder authentication protocol, which will be our running example (see Sec. 2). The ease with which we applied Celf to this example sheds some light on the range of other protocols that could be studied using Celf. Other examples include various encodings of the π-calculus, petri-nets, etc.

CLF is a conservative extension over LF [HHP93], which implies that Celf's functionality is compatible with that of Twelf [PS99]. With a few syntactic modifications Twelf signatures can be read and type checked, and queries can be

* This research has been funded in part by NSF grant CCR-0325808.

executed. Celf does not yet provide any of the meta-theoretic capabilities that sets Twelf apart from its predecessor Elf, such as mode checking, termination checking, coverage checking, and the like, which we leave to future work. In this presentation we concentrate on the two main features of Celf.

Specification. CLF was designed with the objective in mind to simplify the specification of object languages by internalizing common concepts used for specification and make them available to the user. Celf supports dependent types for the encoding of judgments as types, e.g. operational relations between terms and values, open and closed terms, derivability, and logical truth. It also supports the method of *higher-order abstract syntax*, which relieves the user of having to specify substitutions and substitution application. In CLF, every term is equivalent to a unique inductively defined β-normal η-long form modulo α-renaming and let-floating providing an induction principle to reason about the adequacy of the encoding. In addition, CLF provides linear types and concurrency encapsulating monadic types in support of the specification of resource aware and concurrent systems. Examples include operational semantics for languages with effects, transition systems, and protocol stacks.

Experimentation. Celf provides a logic programming interpreter that implements a proof search algorithm for derivations in the CLF type theory in analogy to how Elf implements a logical programming interpreter based on uniform proof search. Celf's interpreter is inspired (with few modifications) by Lollimon [LPPW05], an extension of Lolli, the linear sibling of λ-Prolog. The interpreter implements backward-chaining search within the intuitionistic and linear fragment of CLF and switches to forward-chaining multiset rewriting search upon entering the monad. Celf programs may jump in and out of the concurrency monad and can therefore take advantage of both modes of operation. In addition, the operational semantics of Celf is conservative over the operational semantics of Elf, which means that any Twelf query can also be executed in Celf leading to the same result.

In the remainder of the paper, we illustrate the features of Celf. In Section 2, we describe the Needham-Schroeder protocol followed by a brief overview in Section 3 of the CLF type theory and the protocol specification in Celf [CPWW02b]. Finally, we comment on the implementation and conclude in Section 4.

2 Example

As a small running example, we consider the Needham-Schroeder protocol [NS78]. The protocol serves the authentication of two principals, A and B, and is characterized by the following simplified message exchange

$$A \to B : \{N_a, A\}_{K_b} \tag{1}$$
$$B \to A : \{N_a, N_b\}_{K_a} \tag{2}$$
$$A \to B : \{N_b\}_{K_b} \tag{3}$$

where K_a and K_b are the public keys of A and B, respectively. We write $\{\cdot\}_K$ for the encryption of a message by key K. Two messages may be concatenated using

Kinds

$$K ::= \texttt{type} \mid \texttt{Pi}\ x : A.\ K \qquad \textit{Kinds}$$

Types

$$\begin{array}{rll} A, B & ::= A \texttt{ -o } B \mid \texttt{Pi}\ x : A.\ B \mid A \mathbin{\&} B \mid \langle\top\rangle \mid \{S\} \mid P & \textit{Asynchronous types} \\ P & ::= a \mid P\ N & \textit{Atomic type constructors} \\ S & ::= S_1 \texttt{ * } S_2 \mid 1 \mid \texttt{Exists}\ x : A.\ S \mid A & \textit{Synchronous types} \end{array}$$

Objects

$$\begin{array}{rll} N & ::= \backslash\hat{}\ x.\ N \mid \backslash x.\ N \mid \langle N_1, N_2\rangle \mid \langle\rangle \mid \{E\} \mid & \\ & \quad c \mid x \mid N\ \hat{}N \mid N\ N \mid N\ \texttt{\#}_1 \mid N\ \texttt{\#}_2 & \textit{Objects} \\ E & ::= \texttt{let}\ \{p\} = N\ \texttt{in}\ E \mid M & \textit{Expressions} \\ M & ::= M_1 \texttt{ * } M_2 \mid 1 \mid [N, M] \mid N & \textit{Monadic objects} \\ p & ::= p_1 \texttt{ * } p_2 \mid 1 \mid [x, p] \mid x & \textit{Patterns} \end{array}$$

Signatures

$$\Sigma ::= \cdot \mid a : K.\ \Sigma \mid c : A.\ \Sigma \qquad \textit{Signatures}$$

Fig. 1. Celf syntax

",". N_a and N_b are nonces, randomly generated messages, which are created in line (1) and (2) and compared for identity in line (2) and (3), respectively. We think of A as the initiator of the message exchange and B as the responder. From the point of view of the initiator, two actions are necessary to participate in the protocol.

1. Create a new nonce N_a. Send message $\{N_a, A\}_{K_b}$ (1). Remember B, k_b, and N_a.
2. Recall B, k_b, and N_a. Receive message $\{N'_a, N_b\}_{K_a}$ (2). Check that N_a is identical to N'_a. Send message $\{N_b\}_{K_b}$ (3).

Correspondingly, the responder needs to execute two actions.

1. Receive message $\{N_a, A\}_{K_b}$ (1). Create a new nonce N_b. Send message $\{N_a, N_b\}_{K_a}$ (2). Remember A and N_b.
2. Recall A and N_b. Receive $\{N'_b\}_{K_b}$ (3). Check that N_b is identical to N'_b.

A successful run of the protocol initializes initiator and responder and causes then the initiator to send the first message, the responder to reply, and so forth.

3 Celf

The basis of the Celf system is the CLF type theory [CPWW02a]. The CLF type theory is a dependently typed λ-calculus extended by linear functions, additive and multiplicative pairs, additive and multiplicative units, dependent pairs, and concurrent objects. The syntax of Celf is shown in Fig. 1 and explained below:

Lolli, A `-o` B, is linear implication with linear lambda, `\^`, as introduction form and linear application, `^`, as elimination form.

Pi, `Pi` $x : A.\ B$, is dependent intuitionistic implication, which can also be written A `->` B in the non-dependent case. It has lambda, `\`, and application (juxtaposition) as introduction and elimination forms.

And, A & B, is additive conjunction with $\langle\cdot,\cdot\rangle$ as introduction form and the projections `#`$_i$ as elimination forms.

Tensor or multiplicative conjunction, A `*` B, and *Existential*, `Exists` $x : A.\ S$, are only available inside the monad and can only be deconstructed by the pattern in a `let`-construct. Their introduction forms are tensor, `*`, and dependent pair, $[\cdot,\cdot]$.

Monad, $\{S\}$, is the concurrency monad and represents concurrent computation traces (sequences of `let`-bindings) with a result described by S.

One, 1, is unit for the multiplicative conjunction, and *Top*, $\langle\top\rangle$, is unit for the additive conjunction. Their introduction forms are 1 and $\langle\rangle$ respectively. Combining one with existential quantification allows us to encode the intuitionistic embedding known from linear logic as: $!A =$ `Exists` $x : A.\ 1$. Another common use of one is in the type $\{1\}$ which is the type of concurrent traces in which all linear resources have been consumed. In contrast, $\{\langle\top\rangle\}$ is the type of *any* concurrent trace.

CLF has important meta-theoretical properties including decidability of type-checking and the existence of canonical forms. The notion of definitional equality on CLF terms, types, and kinds is induced by the usual β- and η-rules modulo α-renaming along with one additional rule: Inside the concurrency monad, a sequence of `let`-bindings is allowed to permute as long as the permutation respects the dependencies among bound variables (i.e. permutation is disallowed if it causes a bound variable to escape its scope). This equivalence is the foundation for specifying concurrency in object languages: If a sequence of `let`-bindings represents a concurrent trace of an operational semantics or a protocol communication exchange then CLF will only distinguish between those traces that can lead to observably different results. In other words if two independent events occur within one trace then CLF is completely unaware about the order of their occurrence.

We return to our running example the Needham-Schroeder protocol described in the previous section and illustrate how to specify it in Celf. We follow hereby closely Section 6 of Cervesato et al. [CPWW02b] and refer the interested reader to this technical report for an in depth discussion on how one can derive this encoding and how to reason about its adequacy.

Figure 2 depicts the Celf code specifying the syntactic categories of principals, nonces, public, and private keys in the left column. The right column gives the Celf encoding of messages, where the first two constructors embed principals and nonces into messages, `+` concatenates two messages, and `pEnc` encrypts a message with the public key of principal `A`. Celf's type reconstruction algorithm

```
principal : type.                  msg  : type.
nonce     : type.                  p2m  : principal -> msg.
pubK      : principal -> type.     n2m  : nonce -> msg.
privK     : pubK A -> type.        +    : msg -> msg -> msg.
                                   pEnc : pubK A -> msg -> msg.
```

Fig. 2. Specification of syntactic categories in Celf

```
net    : msg -> type.              init : principal -> type.
rspArg : type.                     resp : principal -> type.
rsp    : rspArg -> type.
```

Fig. 3. Specification of the network, memory, identity in Celf

infers the types of all undeclared uppercase arguments (here `A`), and builds an implicit `Pi`-closure.

The left column of Fig. 3 defines `net` which represents messages being sent on the network. Recall from Section 2 that a principal may need to remember the name of the principal it is trying to authenticate with or specific nonces. In the encoding, the type `rspArg` is used for expressing what the principals can remember, and the type `rsp` for what the principals currently are remembering. In a slight deviation from [CPWW02b] we use two type families `init` and `resp` to assign roles to principals. The corresponding Celf declarations are depicted in the right column of Fig. 3.

What follows below are the two Celf declarations that define initiator and responder of a Needham-Schroeder protocol interaction. The initiator is guarded by `init A` and the responder by `resp B`.

```
nspkInit : init A -o { Exists L : Pi B : principal.
                         pubK B -> nonce -> rspArg.
           Pi B : principal. Pi kB : pubK B.
              { Exists nA : nonce. net (pEnc kB (+ (n2m nA) (p2m A)))
                                 * rsp (L B kB nA) }
         * Pi B : principal. Pi kB : pubK B. Pi kA : pubK A.
           Pi kA' : privK kA. Pi nA : nonce. Pi nB : nonce.
              net (pEnc kA (+ (n2m nA) (n2m nB)))
              -o rsp (L B kB nA)
              -o { net (pEnc kB (n2m nB)) }}.

nspkResp : resp B -o { Exists L : principal -> nonce -> rspArg.
           Pi kB : pubK B. Pi kB' : privK kB.
           Pi A : principal. Pi kA : pubK A. Pi nA : nonce.
              net (pEnc kB (+ (n2m nA) (p2m A)))
              -o { Exists nB : nonce. net (pEnc kA (+ (n2m nA) (n2m nB)))
                                    * rsp (L A nB) }
         * Pi A : principal. Pi kB : pubK B. Pi kB' : privK kB.
           Pi nB : nonce.
              net (pEnc kB (n2m nB)) -o rsp (L A nB) -o { 1 }}.
```

Note that both principals introduce a new parameter `L` to remember the other's identity and their nonce. In addition, the initiator stores the responder's public key to be able to encrypt the second message. The respective nonces are modeled via higher-order abstract syntax, and dynamically created as new and fresh parameters using the `Exists`. Both, messages and memory, are modeled using linear assumptions. Each principal introduces two rules, separated by the top level tensors `*`, which correspond literally to the ones outlined in Section 2. Protocol traces are represented as monadic objects as evidenced by the fact that the declarations end in monadic type `{ · }`.

To experiment with the design in Celf, we use its logic programming engine. For example, in order to find a valid trace of a communication between principal `a`, with public key `ka` and private key `ka'`, and principal `b` with public key `kb` and private key `kb'` we query if it is possible to derive the empty linear context `{1}` from assumptions `init a` and `resp b`. We obtain as answer to the query `#query init a -o resp b -o {1}` the term below, which includes six lets. The first two initiate initiator and responder, and the remaining four correspond to the message exchange of the authentication protocol.

```
Solution: \^ X1. \^ X2. {
    let {[L: Pi B: principal. pubK B -> nonce -> rspArg,
            X3: Pi B: principal. Pi kB: pubK B.
                {Exists nA: nonce. net (pEnc kB (+ (n2m nA) (p2m a))) * rsp (L B kB nA)}
          * X4: Pi B: principal. Pi kB: pubK B. Pi kA: pubK a. privK kA -> Pi nA: nonce.
                Pi nB: nonce. net (pEnc kA (+ (n2m nA) (n2m nB)))
                -o rsp (L B kB nA) -o {net (pEnc kB (n2m nB))}]} = nspkInit ^ X1 in
    let {[L': principal -> nonce -> rspArg,
            X5: Pi kB: pubK b. privK kB -> Pi A: principal. Pi kA: pubK A. Pi nA: nonce.
                net (pEnc kB (+ (n2m nA) (p2m A)))
                -o {Exists nB: nonce. net (pEnc kA (+ (n2m nA) (n2m nB))) * rsp (L' A nB)}
          * X6: Pi A: principal. Pi kB: pubK b. privK kB -> Pi nB: nonce.
                net (pEnc kB (n2m nB)) -o rsp (L' A nB) -o {1}]} = nspkResp ^ X2 in
    let {[nA: nonce, X7: net (pEnc kb (+ (n2m nA) (p2m a))) * X8: rsp (L b kb nA)]}
          = X3 b kb in
    let {[nB: nonce,
            X9: net (pEnc ka (+ (n2m nA) (n2m nB)))
          * X10: rsp (L' a nB)]} = X5 kb kb' a ka nA ^ X7 in
    let {X11: net (pEnc kb (n2m nB))} = X4 b kb ka ka' nA nB ^ X9 ^ X8 in
    let {1} = X6 a kb kb' nB ^ X11 ^ X10 in 1}
```

4 Conclusion

Celf is a system that implements the concurrent logical framework CLF. The implementation includes a type checking, type reconstruction, and proof search algorithm. The implementation employs explicit substitutions, logic variables, and spines and maintains canonical forms through hereditary substitutions.

Celf's type reconstruction algorithm permits programmers to omit inferable top-level `Pi`-quantifiers in declarations of constants. The implicitly bound variables are identified by uppercase names that occur free in declarations. Any use of constants with implicit `Pi`s are then implicitly applied to the correct number of arguments, which are subsequently inferred by Celf via unification.

Free variables in queries play a slightly different role. Uppercase variables occurring free in a query will not be `Pi`-quantified but will instead be considered logic variables and their instantiations are printed for each solution.

The operational semantics of Celf (i.e. the proof search algorithm) works in two modes: when searching for an object of an asynchronous type the algorithm proceeds by a backwards, goal-directed search (resembling pure Prolog), but when searching for an object of a synchronous type the algorithm shifts to an undirected, non-deterministic, and forward-chaining execution, using committed choice instead of backtracking. The latter essentially corresponds to a concurrent multiset rewriting engine.

Both the type reconstruction algorithm and the proof search algorithm rely on logic variables and unification. The implemented unification algorithm works on general CLF terms and handles all relevant aspects: general higher-order terms, linearity, and automatic reordering of bindings inside the monad. For unification problems inside the pattern fragment, which do not have multiple logic variables of monadic type bound by the same sequence of `lets`, the algorithm will always be able to find a most general unifier in case a unifier exists. Any unification problem that falls outside this fragment will be postponed as a constraint, and if it is not resolved by later unifications, it is reported as a leftover constraint. Our empirical experience has shown that this condition characterizes a sufficiently large decidable fragment of higher-order concurrent unification for the application of Celf as a specification and experimentation environment.

References

[CPWW02a] Cervesato, I., Pfenning, F., Walker, D., Watkins, K.: A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Carnegie Mellon University. Department of Computer Science (2002)

[CPWW02b] Cervesato, I., Pfenning, F., Walker, D., Watkins, K.: A concurrent logical framework II: Examples and applications. Technical Report CMU-CS-02-102, Carnegie Mellon University. Department of Computer Science (2002)

[HHP93] Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. Journal of the Association for Computing Machinery 40(1), 143–184 (1993)

[LPPW05] López, P., Pfenning, F., Polakow, J., Watkins, K.: Monadic concurrent linear logic programming. In: Barahona, P., Felty, A.P. (eds.) Proceedings of the 7th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, Lisbon, Portugal, pp. 35–46. ACM Press, New York (2005)

[NS78] Needham, R.M., Schroeder, M.D.: Using encryption for authentication in large networks of computers. Communications of the ACM 21(12), 993–999 (1978)

[PS99] Pfenning, F., Schürmann, C.: System description: Twelf — a meta-logical framework for deductive systems. In: Ganzinger, H. (ed.) CADE 1999. LNCS (LNAI), vol. 1632, pp. 202–206. Springer, Heidelberg (1999)

Canonicity!

Nachum Dershowitz[1,2,⋆]

[1] School of Computer Science, Tel Aviv University, Ramat Aviv, Israel 69978
[2] Microsoft Research, One Microsoft Way, Redmond, WA 98052, USA
nachum.dershowitz@cs.tau.ac.il

Abstract. We describe an abstract proof-theoretic framework based on normal-form proofs, defined using well-founded orderings on proof objects. This leads to robust notions of canonical presentation and redundancy. Fairness of deductive mechanisms – in this general framework – leads to completeness or saturation. The method has so far been applied to the equational, Horn-clause, and deduction-modulo cases.

1 Background

In [18], Knuth invented a *completion* procedure that infers new equations by superposing existing equations with one another (using unification) and also uses equations to simplify one another (by rewriting). When completion terminates successfully, the result is a decision procedure for validity in the theory (variety) of the original equations. In [19], Lankford inaugurated a very fruitful research direction in which superposition is incorporated in a general-purpose theorem-prover for first-order logic.

Eventually, it was noticed [12] that the result of completion is unique – modulo the ordering used for simplification (and for orienting derived equations). In this sense, we can think of the rewrite system produced by completion as being a *canonical* presentation of the given theory, one that provides "cheap" rewrite proofs for all identities of the theory. Similarly, Buchberger's algorithm [7] produces a unique *Gröbner basis*, regardless of nondeterministic choices made along the way.

Huet [17] introduced the notion of *fairness* of completion and showed how fair completion may be viewed as an equational theorem prover. Later, in [1], it was shown how to generalize and formalize equational inference using orderings on proofs, and under what conditions the (finite or infinite) outcome is *complete*, in the sense of providing rewrite proofs of all theorems.

Recently, in a series of papers [13,14,15,3], we proposed quite abstract notions of canonicity and of completion, which can be applied to all manners of inference procedures. Promoting the further study of canonical axiomatizations and their derivation by inference is our goal.

⋆ Research supported in part by the Israel Science Foundation (grant no. 250/05).

2 Theory of Canonicity

In our abstract view of inference, proofs have little structure, but are endowed with two well-founded orderings: a *proof ordering* (under which only proofs with the same conclusion are comparable); and a *subproof ordering*, which is *compatible* with the proof ordering, in the sense that whenever there is a better subproof, there is also a better proof (using the better subproof). By *better*, we mean smaller in the proof ordering; by *good*, we will mean minimal in the ordering.

As usual, every proof has a formula as its conclusion and a set of formulæ as its premises. Theories, in the sense of deductively-closed sets of formulæ, are presumed to obey the standard properties of Tarskian consequence relations (monotonicity, reflexivity, and transitivity).

An *inference procedure* uses some *strategy* in applying a system of (sound) *inference rules*, usually given in the form

$$\frac{A}{c},$$

where c may be any *theorem* of A (that is, the conclusion of any proof with premises from A). We call such rules *expansions.* Expansion rules add lemmata to the growing set of allowable premises.

Many inference procedures also apply *deletion rules* of the form

$$\frac{A, c}{\overline{\overline{}}\,A}$$

– provided that every theorem provable from premises $A \cup \{c\}$ is also provable from A alone (or else completeness would be sacrificed). We are using a double inference line here to indicate that formula c is deleted, *replacing* the set of formulæ above the lines by those below.

In *canonical inference*, deletion is restricted to only allow c to be removed if for every proof with c as a premise, there is a *better* proof without c. We call such restricted deletion steps *contractions.* The point is that such formulæ are truly *redundant* (in the sense of [6]). Once redundant, they will stay redundant; thus, they can be safely removed without endangering completeness of any fairly implemented inference engine.

The following are the basic notions of canonical inference:

1. The *theory* of a presentation (set of formulæ) is the set of all conclusions of proofs using premises from the presentation.
2. A proof is *trivial* if its conclusion is its lone premise.
3. A proof is in *normal form* if it is a good proof when considering the whole theory as potential premises.
4. A presentation is *complete* if it affords at least one normal-form proof for each theorem.
5. A presentation is *saturated* if it supports all normal-form proofs for all theorems.

6. A formula is *redundant* in a presentation, if adding it (or removing it) does not affect normal-form proofs.
7. A presentation is *contracted* (or *reduced*) if it contains no redundant formulæ.
8. A presentation is *perfect* if it is both complete and contracted.
9. A presentation is *canonical* if it is both saturated and contracted.
10. A *critical proof* is a good non-normal-form proof, all of whose (proper) subproofs are in normal form.
11. A formula *persists* in a run of an inference procedure if from some point on it is never deleted.
12. The *result* (in the limit) of a run of an inference procedure is its persistent formulæ.
13. An inference procedure is *fair* if all critical proofs with persistent premises have better proofs at some point.
14. An inference procedure is *uniformly fair* if every trivial normal-form proof is eventually generated.

The following consequences follow from these definitions (see [15,3]):

1. A presentation is contracted if it consists only of premises of normal-form proofs.
2. The smallest saturated presentation is canonical.
3. A presentation is canonical if it consists of all non-redundant formulæ of its theory.
4. A presentation is canonical if it consists of the conclusions (or premises, if you will) of all trivial normal-form proofs.
5. The result of a fair inference procedure is complete.
6. The result of a uniformly fair inference procedure is saturated.
7. The result of an inference procedure is contracted if no redundant formula is allowed to persist.

3 Applications of Canonicity

The abstract approach to inference outlined above has to date been applied to the following situations:

– Ground equations (à la [19,16]) in [13].
– Ground resolution in [15].
– Equational theories (à la [18,17,1]) in [8,9].
– Horn theories (à la [2]) in [5,4].
– Natural deduction in [8].
– Deduction modulo rewriting in [10,11].

Acknowledgements

In this endeavor, I have had the pleasure and privilege of collaborating with Claude Kirchner, Maria Paola Bonacina, and Guillaume Burel.

References

1. Bachmair, L., Dershowitz, N.: Equational inference, canonical proofs, and proof orderings. Journal of the ACM 41, 236–276 (1994), `http://www.cs.tau.ac.il/~nachum/papers/jacm-report.pdf` [viewed May 22, 2008]
2. Bertet, K., Nebut, M.: Efficient algorithms on the Moore family associated to an implicational system. Discrete Mathematics and Theoretical Computer Science 6, 315–338 (2004)
3. Bonacina, M.P., Dershowitz, N.: Abstract canonical inference. ACM Transactions on Computational Logic 8, 180–208 (2007), `http://tocl.acm.org/accepted/240bonacina.pdf` [viewed May 22, 2008]
4. Bonacina, M.P., Dershowitz, N.: Canonical ground Horn theories. Research Report 49/2007, Dipartimento di Informatica, Università degli Studi di Verona (2007), `http://profs.sci.univr.it/~bonacina/papers/TR2007HornCanonicity.pdf` [viewed May 22, 2008]
5. Bonacina, M.P., Dershowitz, N.: Canonical inference for implicational systems. In: Proc. of the Fourth International Joint Conference on Automated Reasoning (IJCAR). LNCS (LNAI), vol. 5195, pp. 380–397. Springer, Heidelberg (2008), `http://www.cs.tau.ac.il/~nachum/papers/CanonicalImplicational.pdf`
6. Bonacina, M.P., Hsiang, J.: Towards a foundation of completion procedures as semidecision procedures. Theoretical Computer Science 146, 199–242 (1995)
7. Buchberger, B.: Ein Algorithmus zum auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal. PhD thesis, Univ. Innsbruck, Austria (1965)
8. Burel, G.: Systèmes canoniques abstraits: Application à la déduction naturelle et à la complétion. Master's thesis, Université Denis Diderot – Paris 7 (2005), `http://www.loria.fr/~burel/download/Burel_Master.pdf` [viewed May 22, 2008]
9. Burel, G., Kirchner, C.: Completion is an instance of abstract canonical system inference. In: Futatsugi, K., Jouannaud, J.-P., Meseguer, J. (eds.) Algebra, Meaning, and Computation. LNCS, vol. 4060, pp. 497–520. Springer, Heidelberg (2006), `http://www.loria.fr/~burel/download/bk4jag.pdf` [viewed May 22, 2008]
10. Burel, G., Kirchner, C.: Cut elimination in deduction modulo by abstract completion. In: Artemov, S., Nerode, A. (eds.) Proc. of the Symposium on Logical Foundations of Computer Science (LFCS). LNCS, pp. 115–131. Springer, Heidelberg (2007)
11. Burel, G., Kirchner, C.: Regaining cut admissibility in deduction modulo using abstract completion (submitted) [viewed May 22, 2008], `http://www.loria.fr/~burel/download/gencomp_ic.pdf`
12. Butler, G., Lankford, D.S.: Experiments with computer implementations of procedures which often derive decision algorithms for the word problem in abstract algebras. Memo MTP-7, Department of Mathematics, Louisiana Tech. University, Ruston, LA (1980)
13. Dershowitz, N.: Canonicity. In: Proc. of the Fourth International Workshop on First-Order Theorem Proving (FTP 2003). Electronic Notes in Theoretical Computer Science, vol. 86, pp. 120–132 (2003), `http://www.cs.tau.ac.il/~nachum/papers/canonicity.pdf` [viewed May 22, 2008]

14. Dershowitz, N., Kirchner, C.: Abstract saturation-based inference. In: Proc. of the Eighteenth IEEE Symposium on Logic in Computer Science, June 2003, pp. 65–74. IEEE Press, Los Alamitos (2003), http://www.cs.tau.ac.il/~nachum/papers/lics2003-final.pdf [viewed May 22, 2008]
15. Dershowitz, N., Kirchner, C.: Abstract canonical presentations. Theoretical Computer Science 357, 53–69 (2006), http://www.cs.tau.ac.il/nachum/papers/AbstractCanonicalPresentations.pdf [viewed May 22, 2008]
16. Gallier, J., Narendran, P., Plaisted, D., Raatz, S., Snyder, W.: An algorithm for finding canonical sets of ground rewrite rules in polynomial time. J. of the Association of Computing Machinery 40, 1–16 (1993)
17. Huet, G.: A complete proof of correctness of the Knuth–Bendix completion algorithm. Journal of Computer and System Sciences 23, 11–21 (1981)
18. Knuth, D.E., Bendix, P.B.: Simple word problems in universal algebras. In: Leech, J. (ed.) Computational Problems in Abstract Algebra, pp. 263–297. Pergamon Press, Oxford (1970)
19. Lankford, D.S.: Canonical inference. Memo ATP-32, Automatic Theorem Proving Project, University of Texas, Austin, TX (1975)

Unification and Matching Modulo Leaf-Permutative Equational Presentations

Thierry Boy de la Tour[1], Mnacho Echenim[1], and Paliath Narendran[2]

[1] CNRS/INPG - Laboratoire d'Informatique de Grenoble, France
thierry.boy-de-la-tour@imag.fr,
mnacho.echenim@imag.fr
[2] Department of Computer Science,
University at Albany — SUNY, Albany, NY 12222, U. S. A.
dran@cs.albany.edu

Abstract. Unification modulo variable-permuting equational presentations is known to be undecidable. We address here the problem of the existence of a unification algorithm for the class of *linear* variable-permuting presentations, also known as *leaf-permutative* presentations. We show that there is none, by exhibiting a leaf-permutative presentation whose unification problem is undecidable. We also exhibit one such presentation whose word and pattern matching problems are PSPACE-complete. The proof proceeds in three stages, by transforming length-preserving string-rewriting systems into *atomic*, then into *bit-swapping* string-rewriting systems, and finally into leaf-permutative presentations.

1 Introduction

Unification, or the problem of solving equations, has nice computational properties when only syntactic equality is considered. As soon as more complex equalities are introduced, starting from congruence modulo associative and commutative functions (or AC theory, see [14,7]), the problem becomes increasingly difficult, leading to a *unification hierarchy*, see [9]. The mere problem of deciding whether an equation admits a solution, rather than computing one, known as *the unification (decision) problem*, spans the complexity scale from linear time to undecidable.

In particular, the AC-unification problem, and the commutative unification problem are NP-complete (see [10,8]). Attempts have been made to define a class of equational theories that would generalize the theories C and AC, and inherit their properties w.r.t. unification. One obvious candidate is the class of *permutative* theories, where identities may only join terms with exactly the same symbols, occurrence for occurrence. But a permutative theory was found in [13] whose unification problem is undecidable.

Since the theory AC can be presented using the axiom

$$(\mathrm{A}') \qquad f(f(x,y),z) \;=\; f(f(z,y),x),$$

together with the axiom of commutativity, in which the left and right-hand sides differ only in their variable positions, the more restricted class of theories axiomatized by *variable-permuting* presentations became of interest. But once again a variable-permuting presentation was found in [11] whose unification problem is undecidable. This presentation however has only non-linear identities.

A further restriction to linear variable-permuting presentations, or *leaf-permutative* as they were called in [1], is therefore natural. A link with permutation groups was exploited in [1] by restricting the congruence to *stratified* terms, thus artificially avoiding overlaps. By restricting overlaps among the axioms, the class of *unify-stable* presentations defined in [6] allows the use of group-theoretic tools with the full congruence, and thus provides a class of leaf-permutative presentations (containing C but not A′C) whose unification problem is in NP. Although the definition of the class is complex, it is still decidable, and even polynomial to decide whether a given presentation admits an equivalent unify-stable presentation, and if so to compute one (see [5]).

We show in this paper that the unification problem on the class of leaf-permutative presentations does not inherit the properties of unify-stable presentations. Indeed, by techniques similar to those of [11] (or to the journal version [12]), we prove that the problem is undecidable (and that the corresponding matching problem is PSPACE-complete), and is therefore as difficult as it is on the entire class of permutative theories.

The paper is organized as follows: Section 2 sets notations and standard definitions. On string-rewriting systems our notations mostly follow [4], and [2] is our reference on equational theories and unification.

In Section 3 we give a brief account of the ideas behind the proofs in [13] and [12]. This allows us to explain the intuition behind the three steps of our own proof. We then proceed with these three steps in Sections 4, 5 and 6. All these results are put together in Section 7.

2 Preliminaries

Any finite alphabet Λ generates a free monoid $\Lambda^\star$, whose elements are Λ-*strings* and whose unit is the *empty string* ε. Any non-empty Λ-string x can be uniquely written as a finite product $x = a_1 \cdots a_n$ of letters $a_i \in \Lambda$, where $n = |x|$ is the *length* of x. For all $1 \leq i \leq j \leq n$ we write $x[i]$ for a_i, and $x[i,j]$ for $a_i \cdots a_j$, which is a *substring* of x. The indices i, j will be referred to as *positions* in x.

A *string-rewriting (or Semi-Thue) system* on a finite alphabet $\Lambda_\mathcal{R}$ is a finite set $\mathcal{R}$ of pairs of $\Lambda_\mathcal{R}$-strings, denoted by $l \to r$ and called *rules* (l is its *left-hand side*, or *lhs* and r its *right-hand side*, or *rhs*). The corresponding *one-step reduction relation* $\to_\mathcal{R}$ is the binary relation on $\Lambda_\mathcal{R}$-strings defined by: $x \to_\mathcal{R} y$ iff there is a rule $l \to r$ in $\mathcal{R}$ and $\Lambda_\mathcal{R}$-strings x_1, x_2 such that $x = x_1 l x_2$ and $y = x_1 r x_2$. The *reduction relation* induced by $\mathcal{R}$ is the reflexive and transitive closure $\to^\star_\mathcal{R}$ of the relation $\to_\mathcal{R}$, and the *Thue-congruence* is the equivalence closure $\leftrightarrow^\star_\mathcal{R}$ of $\to_\mathcal{R}$.

$\mathcal{R}$ is *nœtherian* if there is no infinite sequence $(x_i)_{i\in\mathbb{N}}$ of $\Lambda_\mathcal{R}$-strings such that $x_i \to_\mathcal{R} x_{i+1}$ for all $i \in \mathbb{N}$. $\mathcal{R}$ is *confluent* if, whenever $x \to^\star_\mathcal{R} y$ and $x \to^\star_\mathcal{R} y'$, there exists a $\Lambda_\mathcal{R}$-string z such that $y \to^\star_\mathcal{R} z$ and $y' \to^\star_\mathcal{R} z$. $\mathcal{R}$ is *convergent* if it is both nœtherian and confluent.

A $\Lambda_\mathcal{R}$-string y is an *$\mathcal{R}$-normal-form* if there is no string $z \neq y$ such that $y \to^\star_\mathcal{R} z$; if moreover $x \to^\star_\mathcal{R} y$ then y is an $\mathcal{R}$-normal form *of* x. It is well-known that, if $\mathcal{R}$ is convergent then any $\Lambda_\mathcal{R}$-string has a unique $\mathcal{R}$-normal form.

There is an *overlap* between rules $l \to r$ and $l' \to r'$ either if l is a substring of l', or a non-empty prefix of l is a suffix of l'. When there is no overlap among the rules in a nœtherian system $\mathcal{R}$, it is well-known that $\mathcal{R}$ is confluent (because the problem of deciding which rule to apply vanishes), hence convergent.

A rule $l \to r$ is *length-preserving* if $|l| = |r|$ (which is then called the length of the rule). If all the rules in $\mathcal{R}$ are length-preserving then so is $\mathcal{R}$, and it is obvious that $|x| = |y|$ whenever $x \leftrightarrow^\star_\mathcal{R} y$.

The *word problem* WP($\mathcal{R}$) *for* the string-rewriting system $\mathcal{R}$ on $\Lambda_\mathcal{R}$ is the following:

Instance: a pair of $\Lambda_\mathcal{R}$-strings u, v.
Question: does $u \leftrightarrow^\star_\mathcal{R} v$ hold?

For any three $\Lambda_\mathcal{R}$-strings u, v and w, w is called a *common right-multiplier of u and v for $\mathcal{R}$* if $uw \leftrightarrow^\star_\mathcal{R} vw$ holds. The corresponding computation problem CRMP($\mathcal{R}$) is:

Instance: a pair of $\Lambda_\mathcal{R}$-strings u, v.
Question: does there exist a common right-multiplier of u and v for $\mathcal{R}$?

In the sequel by Σ-terms we mean terms built on a first-order signature Σ and an infinite set of variables V. We will also use the well-known notions of *substitutions*, of a *position p in* a Σ-term t, and of *replacing* the subterm of t *at* position p (denoted $t|_p$) by a term s, yielding $t[s]_p$.

An *equational presentation* is a finite set $\mathcal{E}$ of pairs of Σ-terms, denoted by $s \approx s'$, and called *identities*. The corresponding *one-step reduction relation* $\to_\mathcal{E}$ is the binary relation on Σ-terms defined by: $t \to_\mathcal{E} t'$ iff there is an identity $s \approx s'$ in $\mathcal{E}$, a position p in t and a Σ-substitution μ such that $t|_p = s\mu$ and $t' = t[s'\mu]_p$. The associated *congruence* $\approx_\mathcal{E}$ is the equivalence closure of $\to_\mathcal{E}$.

An identity of the form $s \approx s\sigma$, where s is *linear* (variables occur at most once in s) and σ is a permutation of the variables of s, is called *leaf-permutative*. A presentation is *leaf-permutative* if it only contains leaf-permutative identities. For instance, the presentation A′C in the introduction is leaf-permutative.

We use the same notation WP($\mathcal{E}$) as above for the *word problem for* an equational presentation $\mathcal{E}$:

Instance: a pair of Σ-terms s, t.
Question: does $s \approx_\mathcal{E} t$ hold?

The *equational matching problem modulo* $\mathcal{E}$, or EMP($\mathcal{E}$), is:

INSTANCE: a pair of Σ-terms s, t.
QUESTION: does there exist a substitution μ such that $s\mu \approx_{\mathcal{E}} t$?

The *equational unification problem modulo* $\mathcal{E}$, or EUP($\mathcal{E}$), is:

INSTANCE: a pair of Σ-terms s, t.
QUESTION: does there exist a substitution μ such that $s\mu \approx_{\mathcal{E}} t\mu$?

Computational problems will be compared with respect to the standard Karp-reducibility, i.e. $P \propto Q$ means that there exists a function f from P's instances to Q's instances, such that f can be computed in polynomial time and, for all instance I of P, I is a positive instance of P iff $f(I)$ is a positive instance of Q. We say that f is a *polynomial transformation* from P to Q.

3 Background

In [13] Manfred Schmidt-Schauß observed that all known proofs of the decidable and finitary nature of AC-unification require the use of linear Diophantine equations, and addressed the following problem:

> Does there exist a direct proof of all these properties of AC, and if this is the case, how general is the class of theories where this proof is valid?

He then proceeds by showing that, if this is the case, than this class cannot be as general as the class of permutative theories. This is proved by building a permutative presentation whose unification problem is undecidable. The construction proceeds by transforming the rules of a universal Turing machine U into permutative term rewriting rules.

One key point in this transformation is a clever encoding of tape symbols into terms: the i^{th} tape symbol is represented by the term $t_i = g(0, \ldots, 0, 1, 0, \ldots, 0)$, where the constant 1 appears as the i^{th} argument of g. Thus all symbols are represented using the same number of occurrences of the symbols g, 0 and 1, and therefore replacing a tape symbol by another one is permutative. The same trick is applied for modifying the state of U. Schmidt-Schauß then builds a unification problem modulo the resulting equational theory, from an input string to U, that has a solution if and only if U accepts this input string, which is of course undecidable.

It is then natural to investigate the unification properties of the more restricted class of theories defined by variable-permuting presentations, which still contains AC. This was done by Paliath Narendran and Friedrich Otto in [12], again starting from a universal Turing machine U. The construction requires a number of steps resulting in a length-preserving string-rewriting system $\mathcal{R}$ that simulates U in the following sense: for any input to U we can build a pair of strings u, v which have a common right multiplier w modulo $\mathcal{R}$ if and only if

U accepts this input. The string w represents the finite computation from the input to an accepting state. This proves that CRMP($\mathcal{R}$) is undecidable.

The proof then proceeds by transforming the rules of $\mathcal{R}$ into variable-permuting identities between terms, and the common right multiplier problem into a unification problem (with one variable for w). This uses a trick similar to the one above: the i^{th} letter in the string alphabet is represented by t_i, which is matched uniquely by the term $g(x, \ldots, x, y, x, \ldots, x)$, where the variable y appears as the i^{th} argument of g. These terms are used in the identities, which are therefore variable-permutative. However, the unicity of matching only holds if there are at least three letters. Indeed, if the alphabet is binary then the terms $g(y, x)$ and $g(x, y)$ are unable to discriminate between the two letters represented by $t_1 = g(1, 0)$ and $t_2 = g(0, 1)$.

This is why the construction in [12] is inherently non-linear. But as mentioned in Section 1 AC admits a leaf-permutative presentation, which opens the way to a further restriction of the class of theories. In order to obtain only leaf-permutative rules, the idea in the present paper is to build from $\mathcal{R}$ a length-preserving string-rewriting system on a binary alphabet $\{0, 1\}$. Thus we may drop g and represent the letters 0 and 1 directly by constants 0 and 1. But since any variable can match any letter, we need a further restriction on the string-rewriting rules.

We will see in section 6 that rules that only swap an occurrence of 0 with an occurrence of 1 can be fairly represented by identities between linear terms that only swap two variables x and y. This is basically due to the fact that, if x and y do not match a 0 and a 1, then they match the same digit, so that swapping x with y leaves the corresponding string unaffected. But how can we guarantee such a restricted property on string-rewriting rules?

The answer can be found above: if we replace t_i by t_j, where $i \neq j$, then we essentially swap a 0 with a 1. Reproducing this trick by a suitable representation of letters as binary strings is easy, and is the subject of Section 5. Hence we now only need a string-rewriting system similar to $\mathcal{R}$, except that each rule must change only one letter, so that its binary equivalent swaps only one occurrence of 0 and 1.

We call such a system *atomic*, and we show in the next section how it can be obtained from $\mathcal{R}$. Of course, for each transformation we need to show that the undecidability of the corresponding common right multiplier problem is preserved. In particular this requires to prove that these transformations do not introduce common right multipliers that are not matched by corresponding common right multipliers for $\mathcal{R}$ (i.e. by accepting computations of U).

We will thus establish reductions between common right multiplier problems, and notice that they result from polynomial transformations. This fact is important because we apply the same techniques to the corresponding word problems, at little extra cost. This is used to prove the PSPACE-hardness of the word and matching problems modulo a leaf-permutative theory obtained by applying our transformations not to $\mathcal{R}$ but to a length-preserving string-rewriting system from [3], whose word problem is PSPACE-complete.

4 Atomic String-Rewriting Systems

Definition 1. *A rule $x \to y$ is* atomic *if $|x| = |y|$ and there is a unique index k such that $x[k] \neq y[k]$ i.e. x and y differ by a unique letter. A string-rewriting system is* atomic *when all its rules are atomic.*

Let $\mathcal{R}$ be a length-preserving string-rewriting system. For every rule r in $\mathcal{R}$, of the form $a_1 \cdots a_n \to b_1 \cdots b_n$, we consider n new letters c_i^r for $1 \le i \le n$, and we let $\Lambda_r = \{c_i^r \mid 1 \le i \le n\}$. For all $1 \le i \le n$ let hocus_i^r *denote the rule*

$$c_1^r \cdots c_{i-1}^r a_i a_{i+1} \cdots a_n \;\to\; c_1^r \cdots c_{i-1}^r c_i^r a_{i+1} \cdots a_n$$

and pocus_i^r *denote the rule*

$$b_1 \cdots b_{i-1} c_i^r c_{i+1}^r \cdots c_n^r \;\to\; b_1 \cdots b_{i-1} b_i c_{i+1}^r \cdots c_n^r.$$

Let $\mathrm{a}(r) = \bigcup_{i=1}^{n} \{\mathrm{hocus}_i^r, \mathrm{pocus}_i^r\}$, $\mathrm{a}(\mathcal{R}) = \bigcup_{r \in \mathcal{R}} \mathrm{a}(r)$ *and* $\Lambda_{\mathrm{a}(\mathcal{R})} = \Lambda_{\mathcal{R}} \cup \bigcup_{r \in \mathcal{R}} \Lambda_r$.

Example: if r is the rule $abc \to def$, then we add 3 new letters c_1^r, c_2^r and c_3^r, and $\mathrm{a}(r)$ contains the following rules:

$$abc \xrightarrow{\mathrm{hocus}_1^r} c_1^r bc \xrightarrow{\mathrm{hocus}_2^r} c_1^r c_2^r c \xrightarrow{\mathrm{hocus}_3^r} c_1^r c_2^r c_3^r,$$

$$c_1^r c_2^r c_3^r \xrightarrow{\mathrm{pocus}_1^r} d c_2^r c_3^r \xrightarrow{\mathrm{pocus}_2^r} de c_3^r \xrightarrow{\mathrm{pocus}_3^r} def.$$

The rules hocus_i^r and pocus_i^r are atomic: hocus_i^r replaces a_i by c_i^r, and pocus_i^r replaces c_i^r by b_i. Hence $\mathrm{a}(\mathcal{R})$ is an atomic string rewriting system on $\Lambda_{\mathrm{a}(\mathcal{R})}$, and it is obvious that for any two $\Lambda_{\mathcal{R}}$-strings x and y, if $x \leftrightarrow_{\mathcal{R}}^{\star} y$ then $x \leftrightarrow_{\mathrm{a}(\mathcal{R})}^{\star} y$. In order to prove the converse we need to detect which parts of a $\Lambda_{\mathrm{a}(\mathcal{R})}$-string may not be rewritten from or into a $\Lambda_{\mathcal{R}}$-string.

Definition 2. *Suppose that x is a $\Lambda_{\mathrm{a}(\mathcal{R})}$-string containing a substring $c_i^r \cdots c_j^r$ starting at position k, where $r \in \mathcal{R}$ has length n, and $1 < i \le j < n$. Suppose further that this substring is maximal in the sense that the letter at position $k-1$ in x is not c_{i-1}^r, and the letter at position $k+j-i+1$ is not c_{j+1}^r. Then we say that $c_i^r \cdots c_j^r$ is a* lock at k in x.

A string is lock-free *if it has no lock anywhere.*

Following our Example, let $x = abc_2^r bac_1^r c_2^r$, then c_2^r is a lock at 3 in x, and there are no other locks in x.

When a substring is a lock it is clear that it cannot be rewritten by any rule. Yet applying rules in the neighborhood may change this situation, so we still need to prove that locks are stable under $\mathrm{a}(\mathcal{R})$.

Lemma 1. *For any two $\Lambda_{\mathrm{a}(\mathcal{R})}$-strings x and y such that $x \to_{\mathrm{a}(\mathcal{R})} y$, if one has a lock at k then so does the other, and $x[1, k-1] \leftrightarrow_{\mathrm{a}(\mathcal{R})}^{\star} y[1, k-1]$.*

Proof. Let $c_i^r \cdots c_j^r$ be a lock at k in either x or y, and n be the length of r, so that $1 < i \leq j < n$. First suppose that the lhs of the rule that rewrites x into y overlaps $x[k, j-i+k]$, and therefore that its rhs overlaps $y[k, j-i+k]$. Then there must be an $i \leq l \leq j$ such that c_l^r occurs in the rule. The rule therefore belongs to $\mathrm{a}(r)$, and is either a $\mathrm{hocus}_{i'}^r$ or a $\mathrm{pocus}_{i'}^r$, where $1 \leq i' \leq n$; furthermore, by Definition 1, c_l^r must be the l^{th} letter in the lhs or rhs of the rule. Since c_l^r is at position $l - i + k$ in x or y (see Figure 1 below), the lhs of the rule must be the substring of x starting at position $l - i + k - (l-1) = 1 - i + k$, and ending at position $n - i + k$, i.e. $x[1 - i + k, n - i + k]$, and its rhs must therefore be $y[1 - i + k, n - i + k]$. And therefore the lock $c_i^r \cdots c_j^r$ must be a substring either of the lhs or of the rhs of the rule.

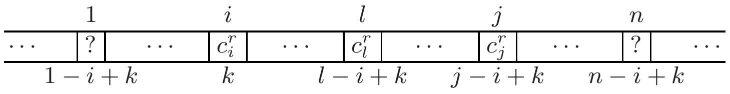


Fig. 1.

If this rule is $\mathrm{hocus}_{i'}^r$ then the lhs is $c_1^r \cdots c_{i'-1}^r a_{i'} \cdots a_n$ and the rhs is $c_1^r \cdots c_{i'}^r a_{i'+1} \cdots a_n$. Since $c_i^r \cdots c_j^r$ is a substring of one of these, it must be a substring of $c_1^r \cdots c_{i'}^r$. This means that $j \leq i'$, hence that $1 \leq i - 1 \leq i' - 1$. The letter c_{i-1}^r therefore occurs on both sides, hence at position $k-1$ in *both* x and y, which contradicts the hypothesis that $c_i^r \cdots c_j^r$ is a lock at k in *either* x or y. If the rule is $\mathrm{pocus}_{i'}^r$ then similarly $i' \leq j < n$ and $x[j+1-i+k] = y[j+1-i+k] = c_{j+1}^r$, which is again impossible.

This proves that the lhs of the rule that rewrites x into y does not overlap $x[k, k+j-i]$, and therefore occurs either entirely to the left of k, so that $x[1, k-1] \rightarrow_{\mathrm{a}(\mathcal{R})} y[1, k-1]$, or entirely to the right of $j-i+k$, so that $x[1, k-1] = y[1, k-1]$; in any case $x[1, k-1] \leftrightarrow^\star_{\mathrm{a}(\mathcal{R})} y[1, k-1]$.

Since the rule does not overlap the lock, this one is unmodified, so that $x[k, j-i+k] = y[k, j-i+k] = c_i^r \cdots c_j^r$. Assume that this is not a lock at k at the same time in x and y. Then by Definition 2 this implies either that exactly one of $x[k-1], y[k-1]$ is equal to c_{i-1}^r, or exactly one of $x[j+1-i+k], y[j+1-i+k]$ is equal to c_{j+1}^r. In both cases, this implies that the lhs of the rule must be $x[1-i+k, n-i+k]$, hence that the rule does overlap the lock, and we have seen that this is impossible. Therefore both x and y have a lock at k (and it is exactly the same). □

A simple induction then shows that:

Corollary 1. *If $x \leftrightarrow^\star_{\mathrm{a}(\mathcal{R})} y$ and x has a lock at k then $x[1, k-1] \leftrightarrow^\star_{\mathrm{a}(\mathcal{R})} y[1, k-1]$.*

Corollary 2. *If $x \leftrightarrow^\star_{\mathrm{a}(\mathcal{R})} y$ then x is lock-free if and only if y is lock-free.*

If a string is lock-free, then by definition its letters from $\Lambda_{\mathrm{a}(\mathcal{R})} \setminus \Lambda_{\mathcal{R}}$ occur either in sequences $c_1^r \cdots c_i^r$ or in sequences $c_j^r \cdots c_n^r$, where $r \in \mathcal{R}$ has length n and $1 \leq i \leq n$, $1 < j \leq n$.

Definition 3. *To any lock-free $\Lambda_{\mathrm{a}(\mathcal{R})}$-string x we associate the $\Lambda_{\mathcal{R}}$-string* $\mathrm{p}(x)$ *obtained from x by replacing every substring $c_1^r \cdots c_i^r$ (with $i \leq n$ maximal) by $a_1 \cdots a_i$, and every substring $c_j^r \cdots c_n^r$ (with $j > 1$ minimal) by $b_j \cdots b_n$, where $r \in \mathcal{R}$ is $a_1 \cdots a_n \to b_1 \cdots b_n$.*

In particular, applying this function on the rhs of the rules hocus_i^r and pocus_r^i, for all $1 \leq i \leq n$, yield:

$$\begin{aligned}\mathrm{p}(c_1^r \cdots c_i^r a_{i+1} \cdots a_n) &= a_1 \cdots a_n,\\ \mathrm{p}(b_1 \cdots b_i c_{i+1}^r \cdots c_n^r) &= b_1 \cdots b_n.\end{aligned}$$

The function p will allow us to transform $\mathrm{a}(\mathcal{R})$-reduction steps between lock-free strings into $\mathcal{R}$-reduction steps, and lock-free common right multipliers for $\mathrm{a}(\mathcal{R})$ into common right multipliers for $\mathcal{R}$. Note that not all lock-free strings rewrite into $\Lambda_{\mathcal{R}}$-strings. Following our Example, the string $c_1^r b c_3^r$ is lock-free, yet cannot be rewritten into or from another string. Here $\mathrm{p}(c_1^r b c_3^r) = abf$ is a $\Lambda_{\mathcal{R}}$-string that may or may not be rewritten into another string. However, all $\mathrm{a}(\mathcal{R})$-reduction steps on lock-free strings can be reflected into $\mathcal{R}$.

Lemma 2. *For any two lock-free $\Lambda_{\mathrm{a}(\mathcal{R})}$-strings x and y, if $x \to_{\mathrm{a}(\mathcal{R})} y$ then either* $\mathrm{p}(x) = \mathrm{p}(y)$ *or* $\mathrm{p}(x) \to_{\mathcal{R}} \mathrm{p}(y)$.

Proof. If a rule hocus_i^r is applied, for some $r \in \mathcal{R}$, then x contains its lhs and y its rhs, and these both translate to the lhs of r by p, hence $\mathrm{p}(x) = \mathrm{p}(y)$. Similarly, if a rule pocus_i^r is applied, with $i > 1$, then both sides translate to the rhs of r, hence $\mathrm{p}(x) = \mathrm{p}(y)$. If the rule pocus_1^r is applied, the lhs translates to the lhs of r, and the rhs to the rhs of r, hence by applying the rule r we get $\mathrm{p}(x) \to_{\mathcal{R}} \mathrm{p}(y)$. □

Corollary 3. *If x and y are lock-free and $x \leftrightarrow^\star_{\mathrm{a}(\mathcal{R})} y$ then* $\mathrm{p}(x) \leftrightarrow^\star_{\mathcal{R}} \mathrm{p}(y)$.

Since $\Lambda_{\mathcal{R}}$-strings are lock-free and invariant under p, this result obviously entails the converse previously mentioned: for any two $\Lambda_{\mathcal{R}}$-strings x and y, if $x \leftrightarrow^\star_{\mathrm{a}(\mathcal{R})} y$ then $x \leftrightarrow^\star_{\mathcal{R}} y$.

Theorem 1. *For any length-preserving string-rewriting system $\mathcal{R}$,*

$$\mathrm{WP}(\mathcal{R}) \propto \mathrm{WP}(\mathrm{a}(\mathcal{R})) \quad \textit{and} \quad \mathrm{CRMP}(\mathcal{R}) \propto \mathrm{CRMP}(\mathrm{a}(\mathcal{R})).$$

Proof. Since for any $\Lambda_{\mathcal{R}}$-strings x and y, $x \leftrightarrow^\star_{\mathcal{R}} y$ iff $x \leftrightarrow^\star_{\mathrm{a}(\mathcal{R})} y$, then the identity function is an obviously polynomial transformation from $\mathrm{WP}(\mathcal{R})$ to $\mathrm{WP}(\mathrm{a}(\mathcal{R}))$, hence $\mathrm{WP}(\mathcal{R}) \propto \mathrm{WP}(\mathrm{a}(\mathcal{R}))$.

Suppose that x, y is a positive instance of the problem $\mathrm{CRMP}(\mathcal{R})$, i.e. there exists a $\Lambda_{\mathcal{R}}$-string z such that $xz \leftrightarrow^\star_{\mathcal{R}} yz$, then $xz \leftrightarrow^\star_{\mathrm{a}(\mathcal{R})} yz$, hence x, y is a positive instance of $\mathrm{CRMP}(\mathrm{a}(\mathcal{R}))$.

Conversely, assume that the pair of $\Lambda_{\mathcal{R}}$-strings x, y is a positive instance of $\mathrm{CRMP}(\mathrm{a}(\mathcal{R}))$, hence there is a $\Lambda_{\mathrm{a}(\mathcal{R})}$-string z such that $xz \leftrightarrow^\star_{\mathrm{a}(\mathcal{R})} yz$. Let $n = |x| = |y|$, and k be the rightmost position in z such that $z' = z[1, k]$ is lock-free. If $k < |z|$ then there is a lock at $k+1$ in z, hence at $n+k+1$ in both xz and

yz, and by Corollary 1 we get $xz' = (xz)[1, n+k] \leftrightarrow^\star_{\mathrm{a}(\mathcal{R})} (yz)[1, n+k] = yz'$. Otherwise $z = z'$, and $xz' \leftrightarrow^\star_{\mathrm{a}(\mathcal{R})} yz'$ is obvious. In both cases xz' and yz' are lock-free, hence by Corollary 3, we get $x\mathrm{p}(z') = \mathrm{p}(xz') \leftrightarrow^\star_{\mathcal{R}} \mathrm{p}(yz') = y\mathrm{p}(z')$, thus $\mathrm{p}(z')$ is a common right multiplier of x and y for $\mathcal{R}$, i.e. x, y is a positive instance of CRMP($\mathcal{R}$). Therefore CRMP($\mathcal{R}$) $\propto$ CRMP(a($\mathcal{R}$)). □

5 Bit-Swapping Systems

Definition 4. *A string, a rule or a string-rewriting system on the alphabet* $\{0, 1\}$ *is called* binary. *A binary rule of the form* $w_1 a w_2 b w_3 \to w_1 b w_2 a w_3$, *where the* w_i*'s are binary strings and* a, b *are two distinct binary digits (i.e.* $\{a, b\} = \{0, 1\}$*) is called* bit-swapping. *A string-rewriting system is* bit-swapping *when it is binary and only contains bit-swapping rules.*

It is obvious that bit-swapping rules are length-preserving, and that the inverse of a bit-swapping rule is also bit-swapping. For instance, the rule $000111 \to 010101$ is bit-swapping.

We now translate an atomic string-rewriting system $\mathcal{A}$ into a bit-swapping string-rewriting system b($\mathcal{A}$), by translating each letter in $\Lambda_{\mathcal{A}}$ into a unique binary string, based on an arbitrary enumeration of $\Lambda_{\mathcal{A}}$. Without loss of generality we assume that $\Lambda_{\mathcal{A}} \cap \{0, 1\} = \emptyset$.

Definition 5. *Let* $\Lambda_{\mathcal{A}} = \{a_1, \ldots, a_n\}$, *and for all* $1 \le i \le n$ *let*

$$\mathrm{b}(a_i) = 0^i\,1\,0^{n+1-i}\,1\,1.$$

We implicitly extend b *to a morphism from the monoid* $\Lambda^\star_{\mathcal{A}}$ *to* $\{0, 1\}^\star$. *Let*

$$\mathrm{b}(\mathcal{A}) \;=\; \{\mathrm{b}(x) \to \mathrm{b}(y) \mid x \to y \in \mathcal{A}\}.$$

We also consider the string-rewriting system $\mathcal{I} = \{\mathrm{b}(a_i) \to a_i \mid 1 \le i \le n\}$ *on the alphabet* $\Lambda_{\mathcal{A}} \uplus \{0, 1\}$.

Note that the b(a_i)'s all have the same length $n + 4$, and if $i < j$ we can write

$$\mathrm{b}(a_i) = 0^i\,1\,0^{j-i-1}\,0\,0^{n+1-j}\,1\,1,$$
$$\mathrm{b}(a_j) = 0^i\,0\,0^{j-i-1}\,1\,0^{n+1-j}\,1\,1.$$

Since $\mathcal{A}$ is atomic, it is thus obvious that b($\mathcal{A}$) is bit-swapping. It is also clear that, for any two $\Lambda_{\mathcal{A}}$-strings x and y, if $x \leftrightarrow^\star_{\mathcal{A}} y$ then $\mathrm{b}(x) \leftrightarrow^\star_{\mathrm{b}(\mathcal{A})} \mathrm{b}(y)$.

It is obvious that $\mathcal{I}$ is nœtherian, since it is length-reducing. Moreover, there is no overlap among the strings b(a_i), hence $\mathcal{I}$ is convergent. This string-rewriting system thus provides a convenient way of defining an inverse to the binary encoding function b, which is required as above for transforming b($\mathcal{A}$)-reduction steps into $\mathcal{A}$-reduction steps.

Definition 6. *For any binary string* u, *we let* q(u) *denote the* $\Lambda_{\mathcal{A}}$*-string obtained by removing the 0's and 1's from the* $\mathcal{I}$*-normal form of* u.

For example, $\mathrm{q}(0010^n1110^n10^{n-1}11) = \mathrm{q}(0a_1 10^{n-2}a_2) = a_1a_2$.

It is obvious that, for any $\Lambda_{\mathcal{A}}$-string x, there is a reduction $\mathrm{b}(x) \rightarrow^{\star}_{\mathcal{I}} x$, and that x, which does not contain any binary digit, is an $\mathcal{I}$-normal form, hence $\mathrm{q}(\mathrm{b}(x)) = x$. Thus q is indeed the inverse of b, but we will need a slightly more general property:

Lemma 3. *If u and v are binary strings and x is a $\Lambda_{\mathcal{A}}$-string, then*

$$\mathrm{q}(u\mathrm{b}(x)v) = \mathrm{q}(u)x\mathrm{q}(v).$$

Proof. Since $u\mathrm{b}(x)v \rightarrow^{\star}_{\mathcal{I}} uxv$, the strings $u\mathrm{b}(x)v$ and uxv have the same $\mathcal{I}$-normal form, hence $\mathrm{q}(u\mathrm{b}(x)v) = \mathrm{q}(uxv)$. Since the lhs of the rules in $\mathcal{I}$ share no letter with x, the $\mathcal{I}$-normal form of uxv is $u'xv'$, where u' and v' are the $\mathcal{I}$-normal forms of u and v respectively. Hence $\mathrm{q}(uxv) = \mathrm{q}(u')\mathrm{q}(x)\mathrm{q}(v')$. Since $\mathrm{q}(u) = \mathrm{q}(u')$, $\mathrm{q}(v) = \mathrm{q}(v')$, and since x does not contain 0 or 1, we have the result. □

From this it is easy to see that our backward translation is compatible with the $\mathcal{A}$-reduction relation:

Lemma 4. *For any binary strings u and v, if $u \rightarrow_{\mathrm{b}(\mathcal{A})} v$ then $\mathrm{q}(u) \rightarrow_{\mathcal{A}} \mathrm{q}(v)$.*

Proof. There exists a rule $x \rightarrow y$ in $\mathcal{A}$, and binary strings u_1 and u_2 such that $u = u_1\mathrm{b}(x)u_2$ and $v = u_1\mathrm{b}(y)u_2$. By Lemma 3 we get $\mathrm{q}(u) = \mathrm{q}(u_1)x\mathrm{q}(u_2)$ and $\mathrm{q}(v) = \mathrm{q}(u_1)y\mathrm{q}(u_2)$. Thus obviously $\mathrm{q}(u) \rightarrow_{\mathcal{A}} \mathrm{q}(v)$. □

Corollary 4. *If $u \leftrightarrow^{\star}_{\mathrm{b}(\mathcal{A})} v$ then $\mathrm{q}(u) \leftrightarrow^{\star}_{\mathcal{A}} \mathrm{q}(v)$.*

In particular, for any two $\Lambda_{\mathcal{A}}$-strings x and y, $\mathrm{b}(x) \leftrightarrow^{\star}_{\mathrm{b}(\mathcal{A})} \mathrm{b}(y)$ is equivalent to $x \leftrightarrow^{\star}_{\mathcal{A}} y$ (since q is the inverse of b). The binary encoding function b thus provides a suitable polynomial transformation:

Theorem 2. *For any atomic string-rewriting system $\mathcal{A}$,*

$$\mathrm{WP}(\mathcal{A}) \propto \mathrm{WP}(\mathrm{b}(\mathcal{A})) \quad \textit{and} \quad \mathrm{CRMP}(\mathcal{A}) \propto \mathrm{CRMP}(\mathrm{b}(\mathcal{A})).$$

Proof. Suppose that the pair of $\Lambda_{\mathcal{A}}$-strings x, y is a positive instance of $\mathrm{CRMP}(\mathcal{A})$. There exists a $\Lambda_{\mathcal{A}}$-string z such that $xz \leftrightarrow^{\star}_{\mathcal{A}} yz$, thus

$$\mathrm{b}(x)\mathrm{b}(z) = \mathrm{b}(xz) \leftrightarrow^{\star}_{\mathrm{b}(\mathcal{A})} \mathrm{b}(yz) = \mathrm{b}(y)\mathrm{b}(z),$$

hence $\mathrm{b}(z)$ is a common right-multiplier of $\mathrm{b}(x)$ and $\mathrm{b}(y)$ for $\mathrm{b}(\mathcal{A})$.

Conversely, assume $\mathrm{b}(x), \mathrm{b}(y)$ is a positive instance of $\mathrm{CRMP}(\mathrm{b}(\mathcal{A}))$, and let w be a binary string such that $\mathrm{b}(x)w \leftrightarrow^{\star}_{\mathrm{b}(\mathcal{A})} \mathrm{b}(y)w$. By Lemma 3 and Corollary 4 we get

$$x\mathrm{q}(w) = \mathrm{q}(\mathrm{b}(x)w) \leftrightarrow^{\star}_{\mathcal{A}} \mathrm{q}(\mathrm{b}(y)w) = y\mathrm{q}(w),$$

hence x, y is a positive instance of $\mathrm{CRMP}(\mathcal{A})$.

The function b is thus an obviously polynomial transformation from $\mathrm{CRMP}(\mathcal{A})$ to $\mathrm{CRMP}(\mathrm{b}(\mathcal{A}))$, hence $\mathrm{CRMP}(\mathcal{A}) \propto \mathrm{CRMP}(\mathrm{b}(\mathcal{A}))$. By taking $z = w = \varepsilon$, the same proof yields $\mathrm{WP}(\mathcal{A}) \propto \mathrm{WP}(\mathrm{b}(\mathcal{A}))$. □

6 Leaf-Permutative Presentations

We now translate rules from any bit-swapping string-rewriting system $\mathcal{B}$ into leaf-permutative identities on terms, by replacing the swapped occurrences 0 and 1 by variables x and y.

Definition 7. *Let f be a binary function symbol, $\Sigma = \{0, 1, f\}$ and V be an infinite set of variables, containing x, y and z. We inductively define a function* trm *from binary strings to the set of Σ-terms on V by:*

$$\mathrm{trm}(\varepsilon) \;=\; z \quad \textit{and} \quad \mathrm{trm}(au) \;=\; f(a, \mathrm{trm}(u)),$$

for any $a \in \{0, 1\}$ and binary string u.

For any rule r in $\mathcal{B}$, of the form $w_1aw_2bw_3 \to w_1bw_2aw_3$ where the w_i's are binary strings and a, b are distinct binary digits, we let

$$\mathrm{T}_r \;=\; \mathrm{trm}(w_1)[z \leftarrow f(x, \mathrm{trm}(w_2)[z \leftarrow f(y, \mathrm{trm}(w_3))])].$$

Finally, we consider the presentation

$$\mathcal{E}_{\mathcal{B}} = \{\mathrm{T}_r \approx \mathrm{T}_r\sigma \mid r \in \mathcal{B}\},$$

where $\sigma = \{x \leftarrow y, y \leftarrow x\}$ is the substitution that swaps x and y.

Note that, for any binary strings u and v, the only variable occurring in the terms $\mathrm{trm}(u)$ and $\mathrm{trm}(v)$ is z, and $\mathrm{trm}(uv) = \mathrm{trm}(u)[z \leftarrow \mathrm{trm}(v)]$.

In order to avoid lots of nested (and messy!) parentheses, we may describe a Σ-term as a binary tree, by drawing left branches vertically and right branches horizontally, as in:

$$\mathrm{trm}(a_1 \cdots a_n) \;=\; \begin{array}{ccccc} f & - \cdots - & f & - & z. \\ | & & | & & \\ a_1 & & a_n & & \end{array}$$

Hence the definition of T_r could be rewritten as

$$\mathrm{T}_r \;=\; \begin{array}{ccccccccccccccccc}
f & -\cdots- & f & - & f & - & f & -\cdots- & f & - & f & - & f & -\cdots- & f & - & z, \\
| & & | & & | & & | & & | & & | & & | & & | & & \\
. & & . & & x & & . & & . & & y & & . & & . & & \\
\multicolumn{3}{c}{\underbrace{\hphantom{f-\cdots-f}}_{w_1}} & & & & \multicolumn{3}{c}{\underbrace{\hphantom{f-\cdots-f}}_{w_2}} & & & & \multicolumn{3}{c}{\underbrace{\hphantom{f-\cdots-f}}_{w_3}} & &
\end{array}$$

which can be compacted to

$$\mathrm{T}_r \;=\; \begin{array}{ccccccccccc} f^\star & - & f & - & f^\star & - & f & - & f^\star & - & z. \\ | & & | & & | & & | & & | & & \\ w_1 & & x & & w_2 & & y & & w_3 & & \end{array}$$

Since σ is a permutation of variables and the terms T_r are linear, the presentation $\mathcal{E}_{\mathcal{B}}$ is leaf-permutative. Note that each term T_r is invariant by reversing the rule r, hence here we consider $\mathcal{B}$ as a Thue system rather than as a semi-Thue system. This of course has no influence on the Thue congruence $\leftrightarrow^\star_{\mathcal{B}}$.

It is easy to prove that bit-swapping $\mathcal{B}$-reduction steps can be reflected as leaf-permutative $\mathcal{E}_{\mathcal{B}}$-reduction steps on the corresponding terms:

Lemma 5. *1. For any rule r in $\mathcal{B}$, of the form $w_1aw_2bw_3 \to w_1bw_2aw_3$, there is a substitution μ such that*

$$\mathrm{trm}(w_1aw_2bw_3) = \mathrm{T}_r\mu \text{ and } \mathrm{trm}(w_1bw_2aw_3) = \mathrm{T}_r\sigma\mu.$$

2. For any two binary strings u and v, if $u \to_{\mathcal{B}} v$ then $\mathrm{trm}(u) \to_{\mathcal{E}_\mathcal{B}} \mathrm{trm}(v)$.

Proof. 1. Let $\mu = \{x \leftarrow a, y \leftarrow b\}$, then

$$\mathrm{trm}(w_1aw_2bw_3) \; = \; \begin{array}{ccccccccccc} f^\star & - & f & - & f^\star & - & f & - & f^\star & - & z \\ | & & | & & | & & | & & | & & \\ w_1 & & a & & w_2 & & b & & w_3 & & \end{array} \; = \; \mathrm{T}_r\mu,$$

and similarly $\mathrm{trm}(w_1bw_2aw_3) = \mathrm{T}_r\{x \leftarrow b, y \leftarrow a\} = \mathrm{T}_r\sigma\mu$.

2. There exist a rule r in $\mathcal{B}$ of the form $w_1aw_2bw_3 \to w_1bw_2aw_3$, and binary strings u', v' such that $u = u'w_1aw_2bw_3v'$ and $v = u'w_1bw_2aw_3v'$. Let $\mu = \{x \leftarrow a, y \leftarrow b\}$ as above, and $\mu' = \mu\{z \leftarrow \mathrm{trm}(v')\}$, so that

$$\mathrm{trm}(u) \; = \; \begin{array}{ccccccccccccc} f^\star & - & f^\star & - & f & - & f^\star & - & f & - & f^\star & - & \mathrm{trm}(v') \\ | & & | & & | & & | & & | & & | & & \\ u' & & w_1 & & a & & w_2 & & b & & w_3 & & \end{array} \; = \; \begin{array}{ccc} f^\star & - & \mathrm{T}_r\mu' \\ | & & \\ u' & & \end{array}$$

If p is the position of z in $\mathrm{trm}(u')$ (i.e. $p = 2^{|u'|}$), then $\mathrm{trm}(u)|_p = \mathrm{T}_r\mu'$. Moreover, it is obvious that $\mathrm{trm}(u)$ and $\mathrm{trm}(v)$ differ only below position p, and that $\mathrm{trm}(v)|_p = \mathrm{T}_r\sigma\mu'$, so that $\mathrm{trm}(v) = \mathrm{trm}(u)[\mathrm{T}_r\sigma\mu']_p$. This shows that $\mathrm{trm}(u)$ rewrites into $\mathrm{trm}(v)$ by applying the rule $\mathrm{T}_r \to \mathrm{T}_r\sigma$ at position p, hence that $\mathrm{trm}(u) \to_{\mathcal{E}_\mathcal{B}} \mathrm{trm}(v)$. □

Corollary 5. *If $u \leftrightarrow^\star_\mathcal{B} v$ then $\mathrm{trm}(u) \approx_{\mathcal{E}_\mathcal{B}} \mathrm{trm}(v)$.*

As in previous sections we need to establish a converse, and hence a backward translation from Σ-terms to binary strings. Since not all terms correspond to strings through trm, we will erase most of the subterms that occur left of the rightmost branch.

Definition 8. *Let δ be the function from Σ-terms on V to $\{0, 1\}$, defined by $\delta(t) = 1$ if $t = 1$, and $\delta(t) = 0$ otherwise. Let* str *be the function from Σ-terms on V to binary strings, inductively defined by:*

$$\begin{cases} \mathrm{str}(0) = \mathrm{str}(1) = \mathrm{str}(v) = \varepsilon \text{ for all } v \in V, \\ \mathrm{str}(f(t,t')) = \delta(t)\mathrm{str}(t') \qquad \text{for all } \Sigma\text{-terms } t, t'. \end{cases}$$

Note that $\delta(t)$ can be considered both as a binary string (of length one) and as a Σ-term, so that we can write $\delta(\delta(t)) = \delta(t)$. The reader can easily check that str is the inverse of trm, i.e. for any binary string u, the string $\mathrm{str}(\mathrm{trm}(u))$ is exactly u. For instance,

$$\mathrm{str}(\mathrm{trm}(010)) \; = \; \mathrm{str}\Big(\begin{array}{ccccccc} f & - & f & - & f & - & z \\ | & & | & & | & & \\ 0 & & 1 & & 0 & & \end{array} \Big) \; = \; \delta(0)\delta(1)\delta(0)\mathrm{str}(z) \; = \; 010.$$

But str can be applied to other terms, such as

$$\mathrm{str}\left(\begin{array}{ccccccc} f & - & f & - & f & - & z \\ | & & | & & | & & \\ f(1,1) & & 1 & & x & & \end{array}\right) \;=\; \delta(f(1,1))\delta(1)\delta(x)\mathrm{str}(z) \;=\; 010.$$

This means that any subterm that does not appear on the first row in the graphical representation, i.e. whose position is not a sequence of 2's, can be replaced by 0 if it is different from 1, without affecting the result of the translation into strings. This is proved in the following lemma, along with another simple property that generalizes the equation

$$\mathrm{str}(\,\mathrm{trm}(u)[z \leftarrow \mathrm{trm}(v)]\,) \;=\; \mathrm{str}(\mathrm{trm}(uv)) \;=\; uv \;=\; \mathrm{str}(\mathrm{trm}(u))\mathrm{str}(\mathrm{trm}(v)).$$

Lemma 6. *For any position p in any term t, if p is a (possibly empty) sequence of 2's then* $\mathrm{str}(t) = \mathrm{str}(t[z]_p)\mathrm{str}(t|_p)$*, otherwise* $\mathrm{str}(t) = \mathrm{str}(t[\delta(t|_p)]_p)$*.*

Proof. We prove these identities by induction on p, but separately. The first is obvious when $p = \varepsilon$, since $\mathrm{str}(t[z]_\varepsilon) = \mathrm{str}(z) = \varepsilon$ and $\mathrm{str}(t|_\varepsilon) = \mathrm{str}(t)$. If $p = 2.q$ is a non-empty sequence of 2's, then t must have the form $f(t_1, t_2)$, hence

$$\begin{aligned}\mathrm{str}(t) &= \delta(t_1)\mathrm{str}(t_2)\\ &= \delta(t_1)\mathrm{str}(t_2[z]_q)\mathrm{str}(t_2|_q) \text{ by the induction hypothesis}\\ &= \mathrm{str}(f(t_1, t_2[z]_q))\mathrm{str}(t_2|_q)\\ &= \mathrm{str}(t[z]_p)\mathrm{str}(t|_p) \text{ since } t|_p = t_2|_q.\end{aligned}$$

We now prove the second identity. We let $a = \delta(t|_p)$ and proceed to prove that $\mathrm{str}(t[a]_p) = \mathrm{str}(t)$. Since p is not empty (it must contain a 1), t is of the form $f(t_1, t_2)$, hence $\mathrm{str}(t) = \delta(t_1)\mathrm{str}(t_2)$.

Suppose p begins with a 1, i.e. there is a position q in t_1 such that $p = 1.q$. If $p = 1$ (the base case) then $\delta(a) = \delta(\delta(t_1)) = \delta(t_1)$, hence

$$\mathrm{str}(t[a]_1) = \mathrm{str}(f(a, t_2)) = \delta(a)\mathrm{str}(t_2) = \delta(t_1)\mathrm{str}(t_2) = \mathrm{str}(t).$$

Otherwise $q \neq \varepsilon$, hence both t_1 and $t_1[a]_q$ have head symbol f, so that both $\delta(t_1)$ and $\delta(t_1[a]_q)$ are equal to 0, and similarly

$$\mathrm{str}(t[a]_p) = \mathrm{str}(f(t_1[a]_q, t_2)) = \delta(t_1[a]_q)\mathrm{str}(t_2) = \delta(t_1)\mathrm{str}(t_2) = \mathrm{str}(t).$$

Suppose now that there is a position q in t_2 such that $p = 2.q$. Then q is not a sequence of 2's, hence we can apply the induction hypothesis on t_2 and q, which yields $\mathrm{str}(t_2) = \mathrm{str}(t_2[\delta(t_2|_q)]_q)$. But $\delta(t_2|_q) = \delta(t|_p) = a$, hence

$$\mathrm{str}(t[a]_p) = \mathrm{str}(f(t_1, t_2[a]_q)) = \delta(t_1)\mathrm{str}(t_2[a]_q) = \delta(t_1)\mathrm{str}(t_2) = \mathrm{str}(t).$$

□

Corollary 6. *For any Σ-term t and binary string u,*

$$\mathrm{str}(\mathrm{trm}(u)[z \leftarrow t]) = u\,\mathrm{str}(t).$$

Proof. Let $t' = \mathrm{trm}(u)[z \leftarrow t]$ and p be the position of z in $\mathrm{trm}(u)$, so that $t'[z]_p = \mathrm{trm}(u)$ and $t'|_p = t$. Since p is a sequence of 2's, we conclude that $\mathrm{str}(t') = \mathrm{str}(\mathrm{trm}(u))\mathrm{str}(t) = u\,\mathrm{str}(t)$. □

We can now prove a converse to Corollary 5. It is obvious that removing subterms requires to remove all reduction steps applied to them. The more subtle point here is how replacing some subterms by 0 does not invalidate those reduction steps that are not removed. This is because these subterms can only be matched by the variables in the identities of $\mathcal{E}_{\mathcal{B}}$.

Lemma 7. *For any two Σ-terms t, t' on V, if $t \rightarrow_{\mathcal{E}_{\mathcal{B}}} t'$ then $\mathrm{str}(t) \leftrightarrow^{\star}_{\mathcal{B}} \mathrm{str}(t')$.*

Proof. If $t \rightarrow_{\mathcal{E}_{\mathcal{B}}} t'$ then by definition there exists a rule r in $\mathcal{B}$, a position p in t and a substitution μ such that

$$t|_p = \mathrm{T}_r\mu \quad\text{and}\quad t' = t[\mathrm{T}_r\sigma\mu]_p.$$

If p is not a sequence of 2's, then by Lemma 6 $\mathrm{str}(t) = \mathrm{str}(t[\delta(\mathrm{T}_r\mu)]_p)$ and $\mathrm{str}(t') = \mathrm{str}(t[\delta(\mathrm{T}_r\sigma\mu)]_p)$. But the head symbol of T_r is f, hence $\delta(\mathrm{T}_r\mu) = \delta(\mathrm{T}_r\sigma\mu) = 0$, and therefore $\mathrm{str}(t) = \mathrm{str}(t[0]_p) = \mathrm{str}(t')$, so that $\mathrm{str}(t) \leftrightarrow^{\star}_{\mathcal{B}} \mathrm{str}(t')$ is trivially true.

Otherwise p is a sequence of 2's, and by Lemma 6 $\mathrm{str}(t) = \mathrm{str}(t[z]_p)\mathrm{str}(\mathrm{T}_r\mu)$ and $\mathrm{str}(t') = \mathrm{str}(t'[z]_p)\mathrm{str}(\mathrm{T}_r\sigma\mu)$. But $t[z]_p = t'[z]_p$, hence $\mathrm{str}(t)$ and $\mathrm{str}(t')$ share the same prefix. The term T_r contains the three variables x, y, z, and r has the form $w_1 a w_2 b w_3 \rightarrow w_1 b w_2 a w_3$, where $\{a, b\} = \{0, 1\}$, thus

$$\mathrm{T}_r\mu \;=\; \begin{array}{ccccccccccc} f^{\star} & - & f & - & f^{\star} & - & f & - & f^{\star} & - & z\mu. \\ | & & | & & | & & | & & | & & \\ w_1 & & x\mu & & w_2 & & y\mu & & w_3 & & \end{array}$$

A simple induction shows that

$$\begin{aligned} \mathrm{str}(\mathrm{T}_r\mu) &= w_1\,\delta(x\mu)\,w_2\,\delta(y\mu)\,w_3\,\mathrm{str}(z\mu), \\ \mathrm{str}(\mathrm{T}_r\sigma\mu) &= w_1\,\delta(y\mu)\,w_2\,\delta(x\mu)\,w_3\,\mathrm{str}(z\mu). \end{aligned}$$

If $\delta(x\mu) = \delta(y\mu)$ then these two terms are identical, hence so are $\mathrm{str}(t)$ and $\mathrm{str}(t')$, so that $\mathrm{str}(t) \leftrightarrow^{\star}_{\mathcal{B}} \mathrm{str}(t')$ is trivial. If $\delta(x\mu) \neq \delta(y\mu)$ then these are two distinct binary digits, hence $\{\delta(x\mu), \delta(y\mu)\} = \{0, 1\} = \{a, b\}$. Therefore $\delta(x\mu)$ is either equal to a, whence $\mathrm{str}(\mathrm{T}_r\mu) \rightarrow_{\mathcal{B}} \mathrm{str}(\mathrm{T}_r\sigma\mu)$, or it is equal to b, and $\mathrm{str}(\mathrm{T}_r\sigma\mu) \rightarrow_{\mathcal{B}} \mathrm{str}(\mathrm{T}_r\mu)$. In both cases $\mathrm{str}(t) \leftrightarrow^{\star}_{\mathcal{B}} \mathrm{str}(t')$. □

Corollary 7. *If $t \approx_{\mathcal{E}_{\mathcal{B}}} t'$ then $\mathrm{str}(t) \leftrightarrow^{\star}_{\mathcal{B}} \mathrm{str}(t')$.*

The following reduction from the common right-multiplier to the unification problem is similar to the one established in [12].

Theorem 3. *For any bit-swapping string-rewriting system $\mathcal{B}$,*

$$\mathrm{WP}(\mathcal{B}) \propto \mathrm{WP}(\mathcal{E}_{\mathcal{B}}),\ \ \mathrm{CRMP}(\mathcal{B}) \propto \mathrm{EUP}(\mathcal{E}_{\mathcal{B}}) \ \ \textit{and}\ \ \mathrm{WP}(\mathcal{B}) \propto \mathrm{EMP}(\mathcal{E}_{\mathcal{B}}).$$

Proof. Let u and v be two binary strings, and suppose that the pair u, v is a positive instance of $\mathrm{WP}(\mathcal{B})$, then by Corollary 5 the pair $\mathrm{trm}(u), \mathrm{trm}(v)$ is a positive instance of $\mathrm{WP}(\mathcal{E}_{\mathcal{B}})$. Conversely, if $\mathrm{trm}(u) \approx_{\mathcal{E}_{\mathcal{B}}} \mathrm{trm}(v)$ then by Corollary 7 $u = \mathrm{str}(\mathrm{trm}(u)) \leftrightarrow^{\star}_{\mathcal{B}} \mathrm{str}(\mathrm{trm}(v)) = v$. The transformation from u, v to $\mathrm{trm}(u), \mathrm{trm}(v)$ is polynomial, hence $\mathrm{WP}(\mathcal{B}) \propto \mathrm{WP}(\mathcal{E}_{\mathcal{B}})$.

Suppose now that u, v is a positive instance of $\mathrm{CRMP}(\mathcal{B})$. Then there is a binary string w such that $uw \leftrightarrow^{\star}_{\mathcal{B}} vw$, hence $\mathrm{trm}(uw) \approx_{\mathcal{E}_{\mathcal{B}}} \mathrm{trm}(vw)$ by Corollary 5, which can be written

$$\mathrm{trm}(u)[z \leftarrow \mathrm{trm}(w)] \approx_{\mathcal{E}_{\mathcal{B}}} \mathrm{trm}(v)[z \leftarrow \mathrm{trm}(w)],$$

and therefore the substitution $\{z \leftarrow \mathrm{trm}(w)\}$ is an $\mathcal{E}_{\mathcal{B}}$-unifier of $\mathrm{trm}(u)$ and $\mathrm{trm}(v)$. Conversely, suppose that $\mathrm{trm}(u)$ and $\mathrm{trm}(v)$ are $\mathcal{E}_{\mathcal{B}}$-unifiable, then there is a term t such that $\mathrm{trm}(u)[z \leftarrow t] \approx_{\mathcal{E}_{\mathcal{B}}} \mathrm{trm}(v)[z \leftarrow t]$. Hence by Corollary 7 $\mathrm{str}(\mathrm{trm}(u)[z \leftarrow t]) \leftrightarrow^{\star}_{\mathcal{B}} \mathrm{str}(\mathrm{trm}(v)[z \leftarrow t])$, so that $u\,\mathrm{str}(t) \leftrightarrow^{\star}_{\mathcal{B}} v\,\mathrm{str}(t)$ holds by Corollary 6, i.e. $\mathrm{str}(t)$ is a common right multiplier of u and v for $\mathcal{B}$. The function trm is therefore a polynomial transformation from $\mathrm{CRMP}(\mathcal{B})$ to $\mathrm{EUP}(\mathcal{E}_{\mathcal{B}})$.

We now transform an instance u, v of the binary word problem for $\mathcal{B}$ in a slightly more elaborate way: we let u' denote the longest string among u and v, and v' the other one. If u, v is a positive instance of $\mathrm{WP}(\mathcal{B})$ then $u' \leftrightarrow^{\star}_{\mathcal{B}} v'$ (and of course they have the same length), hence by Corollary 5 $\mathrm{trm}(u') \approx_{\mathcal{E}_{\mathcal{B}}} \mathrm{trm}(v')$. Thus obviously $\mu = \{z \leftarrow 0\}$ satisfies the relation $\mathrm{trm}(u')\mu \approx_{\mathcal{E}_{\mathcal{B}}} \mathrm{trm}(v')[z \leftarrow 0]$, i.e. $\mathrm{trm}(u'), \mathrm{trm}(v')[z \leftarrow 0]$ is a positive instance of $\mathrm{EMP}(\mathcal{E}_{\mathcal{B}})$.

Conversely, suppose that $\mathrm{trm}(u'), \mathrm{trm}(v')[z \leftarrow 0]$ is a positive instance of $\mathrm{EMP}(\mathcal{E}_{\mathcal{B}})$, then there is a Σ-term t such that $\mathrm{trm}(u')[z \leftarrow t] \approx_{\mathcal{E}_{\mathcal{B}}} \mathrm{trm}(v')[z \leftarrow 0]$. Corollaries 6 and 7 yield

$$u'\mathrm{str}(t) = \mathrm{str}(\mathrm{trm}(u')[z \leftarrow t]) \leftrightarrow^{\star}_{\mathcal{B}} \mathrm{str}(\mathrm{trm}(v')[z \leftarrow 0]) = v'\mathrm{str}(0) = v'.$$

But $\mathcal{B}$ is length-preserving, and $|v'| \leq |u'|$, hence $\mathrm{str}(t)$ must be the empty string, so that $u' \leftrightarrow^{\star}_{\mathcal{B}} v'$, and u, v is therefore a positive instance of $\mathrm{WP}(\mathcal{B})$. The transformation from u, v to $\mathrm{trm}(u'), \mathrm{trm}(v')[z \leftarrow 0]$ is obviously polynomial, hence $\mathrm{WP}(\mathcal{B}) \propto \mathrm{EMP}(\mathcal{E}_{\mathcal{B}})$. □

7 Conclusion

We can now reach the conclusion by applying our transformations to length-preserving string-rewriting systems that are known to possess specific computational properties.

Theorem 4. *There is a leaf-permutative presentation whose word and equational matching problems are PSPACE-complete, and there is one whose equational unification problem is undecidable.*

Proof. It is obvious from Theorems 1, 2 and 3 that, for any length-preserving string-rewriting system $\mathcal{R}$ there exists a leaf-permutative equational presentation $\mathcal{E} = \mathcal{E}_{\mathrm{b}(\mathrm{a}(\mathcal{R}))}$ such that

$$\mathrm{WP}(\mathcal{R}) \propto \mathrm{WP}(\mathcal{E}), \; \mathrm{WP}(\mathcal{R}) \propto \mathrm{EMP}(\mathcal{E}) \;\text{ and }\; \mathrm{CRMP}(\mathcal{R}) \propto \mathrm{EUP}(\mathcal{E}).$$

It was proved in [3] (Theorem 2.3 p. 528) that there is a length-preserving string-rewriting system whose word problem is PSPACE-complete, hence the word and equational matching problems of the corresponding leaf-permutative presentation $\mathcal{E}$ are PSPACE-hard. Moreover, the matching problem can be solved in PSPACE for all leaf-permutative presentation (as for all variable-permuting presentations, see [12]), and the same obviously holds for the word problem, hence EMP($\mathcal{E}$) and WP($\mathcal{E}$) are both PSPACE-complete.

It was proved in [12] (Theorem 3.4 p. 96) that there is a length-preserving string-rewriting system whose common right multiplier problem is undecidable, hence the equational unification problem of the corresponding leaf-permutative presentation is obviously undecidable. □

References

1. Avenhaus, J., Plaisted, D.: General algorithms for permutations in equational inference. Journal of Automated Reasoning 26, 223–268 (2001)
2. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1998)
3. Bauer, G., Otto, F.: Finite complete rewriting systems and the complexity of the word problem. Acta Informatica 21 (1984)
4. Book, R.V., Otto, F.: String-rewriting systems. Springer, Heidelberg (1993)
5. Boy de la Tour, T., Echenim, M.: Determining unify-stable presentations. In: Baader, F. (ed.) RTA 2007. LNCS, vol. 4533, Springer, Heidelberg (2007)
6. Boy de la Tour, T., Echenim, M.: Permutative rewriting and unification. Information and Computation 25(4), 624–650 (2007)
7. Fages, F.: Associative-commutative unification. JSC 3(3), 257–275 (1987)
8. Garey, M., Johnson, D.S.: Computers and intractability: a guide to the theory of NP-completeness. W.H. Freeman, San Francisco (1979)
9. Siekmann, J.: Unification theory. Journal of Symbolic Computation 7, 207–274 (1989)
10. Kapur, D., Narendran, P.: Complexity of unification problems with associative-commutative operators. Journal of Automated Reasoning 9(2), 261–288 (1992)
11. Narendran, P., Otto, F.: Some results on equational unification. In: Stickel, M.E. (ed.) CADE 1990. LNCS, vol. 449, pp. 276–291. Springer, Heidelberg (1990)
12. Narendran, P., Otto, F.: Single versus simultaneous equational unification and equational unification for variable-permuting theories. Journal of Automated Reasoning 19(1), 87–115 (1997)
13. Schmidt-Schauß, M.: Unification in permutative equational theories is undecidable. Journal of Symbolic Computation 8(4), 415–421 (1989)
14. Stickel, M.E.: A unification algorithm for associative-commutative functions. Journal of the ACM 28(2), 423–434 (1981)

Modularity of Confluence
Constructed

Vincent van Oostrom

Utrecht University, Department of Philosophy
Heidelberglaan 8, 3584 CS Utrecht, The Netherlands
`Vincent.vanOostrom@phil.uu.nl`

Abstract. We present a novel proof of Toyama's famous modularity of confluence result for term rewriting systems. Apart from being short and intuitive, the proof is modular itself in that it factors through the decreasing diagrams technique for abstract rewriting systems, is constructive in that it gives a construction for the converging rewrite sequences given a pair of diverging rewrite sequences, and general in that it extends to opaque constructor-sharing term rewriting systems. We show that for term rewrite systems with extra variables, confluence is not preserved under *de*composition, and discuss whether for these systems confluence *is* preserved under composition.

1 Introduction

We present a novel proof of the classical result due to Toyama, that confluence is a modular property of term rewriting systems, that is, the disjoint union of two term rewriting systems is confluent if and only if both systems are confluent.

To illustrate both our proof method and its difference with existing proofs in the literature [1,2,3], we make use of the following example.

Example 1. Let $\mathcal{S}$ be the disjoint union $\mathcal{CL} \uplus \mathcal{E}$ of the term rewriting systems $\mathcal{CL} = (\{@, I, K, S\}, \{Ix \to x, Kxy \to x, Sxyz \to xz(yz)\})$ for combinatory logic (where @ is left implicit and associates to the left) and $\mathcal{E} = (\{*, a, b\}, \{x * x \to x, a \to b\})$ (with $*$ written infix).

Both these term rewriting systems are confluent, $\mathcal{CL}$ is since it is left-linear and non-overlapping (Rosen, see [4, Sect. 4.1]) and $\mathcal{E}$ is since it is terminating and all its critical pairs (none) are convergent (Huet, see [4, Lemma 2.7.15]).

Now consider the following peak in $\mathcal{S}$ for arbitrary $\mathcal{CL}$-terms t, s, u with $t \to_{\mathcal{CL}} u$:

$$(u * s)a \leftarrow_{\mathcal{CL}} (t * s)a \to_{\mathcal{E}} (t * s)b$$

Intuitively, it is easy to find a common reduct:

$$(u * s)a \to_{\mathcal{E}} (u * s)b \leftarrow_{\mathcal{CL}} (t * s)b$$

and indeed this valley is constructed by our proof. We will now informally explain our proof using classical modularity terminology from the papers mentioned above, which also can be found in the textbooks [5,4].

Our proof is based on a novel decomposition of a term into its *base vector* and its *base*. These notions will be defined formally in Section 3, but can be understood as the vector of aliens (also known as principal subterms) of *maximal* rank of the term, and its context, respectively. This decomposition of terms in turn induces a decomposition of reductions on base vectors and bases such that, roughly speaking, both are confluent and they commute.

Example 2. The source $(t * s)a$ of the initial peak has two aliens $t * s$ and a, only the first of which is of maximal rank 2. Accordingly, its base vector consists only of the former so is $(t * s)$, and its base is $(x_1)a$. Indeed the base vector and base compose, by substituting the former for x_1 in the latter, to the original term.

For the peak, the step on the left decomposes as a step on the base vector: $(u * s) \leftarrow (t * s)$ within base $(x_1)a$, whereas the step on the right decomposes as a step on the base: $(x_1)a \to (x_1)b$ with base vector $(t * s)$.

These steps commute, giving rise to the valley on the previous page, consisting of two steps. The step on the left is composed of the step on the base: $(x_1)a \to (x_1)b$ and base vector $(u * s)$, and the step on the right is composed of the step on the base vector: $(u * s) \leftarrow (t * s)$ and base $(x_1)b$.

Focussing attention on aliens of *maximal* rank is in a sense dual to the idea of the proofs in the literature [1,2,3], which are based on representing aliens by terms of *minimal* rank. To that end each of the latter three proofs relies on a test whether a layer may collapse or not (*root preservedness*, *inner preservedness*, and *cap-stability*, respectively). As the test whether a layer may collapse or not is an undecidable property, the proofs in the literature are non-constructive.

Example 3. To construct the common reduct for the peak above, each of these proofs depends on testing whether the term $t * s$ may collapse, i.e. rewrite to a $\mathcal{CL}$-term, or not. Since the rule for $*$ is non-left-linear, this requires testing whether t and s have a common reduct in $\mathcal{CL}$, which is an undecidable property.

In contrast, our proof yields modularity of *constructive* confluence: a pair of converging rewrite sequences can be constructed for every pair of diverging rewrite sequences, in the disjoint union of two term rewriting systems if and only if the same holds for each of the systems separately.

Example 4. $\mathcal{CL}$ is constructively confluent since a valley can be constructed for a given peak via the so-called orthogonal projections ([4, Chapter 8]), and $\mathcal{E}$ is constructively confluent since a valley can be constructed for a given peak by termination ([4, Chapter 6]). By our main result (Corollary 1) their disjoint union $\mathcal{S}$ is constructively confluent.

After recapitulating the relevant notions from rewriting in Section 2, our proof of modularity of confluence is presented in Section 3, and argued to be constructive in Section 4. In Section 5, we discuss (im)possible extensions of our technique.

2 Preliminaries

We recapitulate from the literature standard rewriting notions pertaining to this paper. The reader is referred to the original papers [1,2,3] for more background on modularity of confluence (Theorem 2) and to [6] for the confluence by decreasing diagrams technique (Theorem 1). As the textbooks [5,4] give comprehensive overviews of both topics, we have based our set-up on them, and we refer to them for the standard notions employed. The novel part of these preliminaries starts with Definition 1.

Rewriting preliminaries We employ arrow-like notations such as $\to$, $\rightsquigarrow$, $\triangleright$, $\blacktriangleright$ to range over *rewrite* relations which are binary relations on a set of *objects*. The *converse* $\to^{-1}$ of a rewrite relation $\to$ is also denoted by mirroring the symbol $\leftarrow$, its reflexive closure is denoted by $\to^{=}$ and its transitive closure by $\to^{+}$. Replicating a symbol is used to denote an idea of repetition; the replication $\twoheadrightarrow$ of $\to$ will be used simply as another notation for its reflexive–transitive closure $\to^{*}$; the meanings of $\triangleright\!\triangleright$ and $\blacktriangleright\!\blacktriangleright$ will be given below. A *peak* (*valley*) is a witnesses to $\twoheadleftarrow;\twoheadrightarrow$ ($\twoheadrightarrow;\twoheadleftarrow$), i.e. a pair of diverging (converging) rewrite sequences. A rewrite relation $\to$ is *confluent* $\mathrm{Con}(\to)$ if every peak can be completed by a valley, i.e. $\twoheadleftarrow ; \twoheadrightarrow \,\subseteq\, \twoheadrightarrow ; \twoheadleftarrow$, where ; and $\subseteq$ denote relation composition and set inclusion respectively, viewing relations as sets of ordered pairs.

Theorem 1 (Decreasing Diagrams [6]). *Let* $\to\, = \bigcup_{\ell \in L} \to_\ell$ *where* L *is a set of* labels *with a terminating transitive relation* $\succ$. *If it holds that* $\leftarrow_\ell ; \to_k \,\subseteq\, \twoheadrightarrow_{\curlyvee \ell} ; \to_k^{=} ; \twoheadrightarrow_{\curlyvee \{\ell,k\}} ; \twoheadleftarrow_{\curlyvee \{k,\ell\}} ; \leftarrow_\ell^{=} ; \twoheadleftarrow_{\curlyvee k}$ *for all* ℓ,k, *then* $\to$ *is confluent. Here* $\curlyvee K = \{j \mid \exists i \in K, i \succ j\}$ *and* $\curlyvee j = \curlyvee \{j\}$.

Term rewriting preliminaries A *term* rewriting system (TRS) is a system (Σ, R) with Σ its signature and R its set of rules. A *signature* is a set of *(function) symbols* with each of which a natural number, its *arity*, is associated. A *rule* is a pair (l, r), denoted by $l \to r$, of terms over the signature extended with an implicit set of nullary *(term) variables*. For a term of shape $a(\boldsymbol{t})$, a is its *head-*symbol, which can be a variable or function symbol, and terms among $\boldsymbol{t}$ are its *direct subterms*. A *subterm* of a term is either the term itself, or a subterm of a direct subterm. Witnessing this inductive notion of subterm yields the notion of *occurrence* of a subterm *in* a term. As usual, we require that the head-symbol of the *left-hand side* l of a rule $l \to r$ not be a variable, and that the variables occurring in the *right-hand side* r occur in l. A rule is *left-linear* if variables occur at most once in its left-hand side, and *collapsing* if the head-symbol of its right-hand side is a variable. For a given TRS $\mathcal{T} = (\Sigma, R)$, its associated rewrite relation $\to_{\mathcal{T}}$ is the binary relation on terms over Σ defined by $t \to_{\mathcal{T}} s$ if $t = C[l^{\tau}]$ and $C[r^{\tau}] = s$ for some single-hole context C, rule $l \to r \in R$, and substitution τ. A *substitution* is a homomorphism on terms induced by the variables, and a *context* is a term over Σ extended with the *hole* $\Box$ (which will technically be treated as a fresh nameless variable). The application of a substitution τ to the term t is denoted by t^{τ}. The main operation on contexts is *hole filling*, i.e. replacing occurrences of the hole by terms. If C is a context and $\boldsymbol{t}$ a term vector with *length* $|\boldsymbol{t}|$ equal to the number of holes in C, then $C[\boldsymbol{t}]$ denotes filling the holes in C from left to right with $\boldsymbol{t}$. Properties of rewrite relations apply to TRSs $\mathcal{T}$ via $\to_{\mathcal{T}}$. Below, we fix TRSs $\mathcal{T}_i = (\Sigma_i, R_i)$ and their associated rewrite relations $\to_i$, for $i \in \{1, 2\}$.

Modularity preliminaries With $\mathcal{T}_1 \uplus \mathcal{T}_2 = (\Sigma, R)$ we denote the disjoint union of $\mathcal{T}_1$ and $\mathcal{T}_2$, where $\Sigma = \Sigma_1 \uplus \Sigma_2$ and $R = R_1 \uplus R_2$. Its associated rewrite relation is denoted by $\rightarrow$. A property P is *modular* if $P(\mathcal{T}_1 \uplus \mathcal{T}_2) \iff P(\mathcal{T}_1) \,\&\, P(\mathcal{T}_2)$.

Theorem 2 (Modularity of Confluence [1]). *$Con(\rightarrow) \iff Con(\rightarrow_1)$ & $Con(\rightarrow_2)$.*

A term t over Σ can be uniquely written as $C[\boldsymbol{t}]$ such that C is a non-empty context over one of the signatures Σ_1,Σ_2, and the head-symbols of the term vector $\boldsymbol{t}$ all *not* belong to that signature, where we let variables belong to both signatures; this situation is denoted by $C[\![\boldsymbol{t}]\!]$ and C and $\boldsymbol{t}$ are said to be the *top* and *alien* vector of t, respectively. The *rank* of t is 1 plus the maximum of the ranks of $\boldsymbol{t}$ and 0. A step $t \rightarrow s$ is *top*-collapsing if it is generated by a collapsing rule in the empty context and the head-symbol of s does not belong to the signature the head-symbol of t belongs to. Observe that the rank of t is then greater than the rank of s. For vectors of terms $\boldsymbol{t}$, $\boldsymbol{s}$ of the same length $|\boldsymbol{t}| = n = |\boldsymbol{s}|$, we write $\boldsymbol{t} \propto \boldsymbol{s}$ if $t_i = t_j$ entails $s_i = s_j$, for all $1 \leq i, j \leq n$. For referring to notions pertaining to the components of $\mathcal{T}_1 \uplus \mathcal{T}_2$, *colors* $c, b, w \in \{1, 2\}$ are employed. We refer to b and w as *black* and *white*, respectively, implicitly assuming they are distinct, i.e. $b = 3 - w$.

To illustrate our concepts and proof, we use the following running example.

Example 5. Let $\mathcal{T}$ be the disjoint union $\mathcal{T}_1 \uplus \mathcal{T}_2$ of the term rewriting systems $\mathcal{T}_1 = (\{a, f\}, \{f(x, x) \rightarrow x\})$ and $\mathcal{T}_2 = (\{I, J, K, G, H\}, \{G(x) \rightarrow I, I \rightarrow K, G(x) \rightarrow H(x), H(x) \rightarrow J, J \rightarrow K\})$. The goal will be to transform the peak

$$f(I, H(a)) \leftarrow f(I, G(a)) \leftarrow f(G(a), G(a)) \rightarrow G(a)$$

into a valley. For $b = 1$, the term $f(I, G(a))$ in this peak is top-black, has top

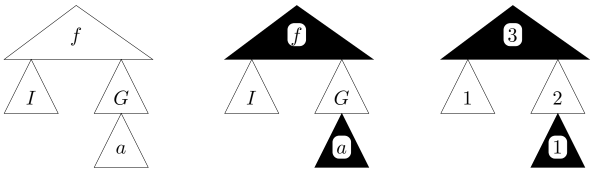


Fig. 1. Ranking a term

$f(\Box, \Box)$, alien vector $(I,G(a))$ and rank 3, since $G(a)$ is top-white, has top $G(\Box)$, alien vector (a) and rank 2, and I, a have no aliens so have rank 1 (see Figure 1).

Definition 1. *For a given rank r and color c, c-*rank-r *terms are terms either of rank r and top-c and then called* tall*, or of rank lower than r then called* short*.*

Every term of rank r is a c-rank-q term for *every* c and $q > r$, and a c-rank-r term for *some* color c. The aliens of a c-rank-$(r+1)$ term are $(3-c)$-rank-r-terms, and if tall its top has color c (the top of short terms can be of either color).

Example 6. The term $f(I, G(a))$ in the peak in Example 5, is short c-rank-r for *every* c and $r > 3$. Although the term is tall black-rank-3, it is *not* white-rank-3.

Rewriting a term does not increase its rank [5, p. 264][4, Proposition 5.7.6] by the constraints on the left- and right-hand sides of TRS rules, and it easily follows:

Proposition 1. *c-rank-r terms are closed under rewriting.*

3 Modularity of Confluence, by Decreasing Diagrams

We show modularity of confluence (Theorem 3) based on the decomposition of terms into bases and vectors of tall aliens, and of steps into tall and short ones, as outlined in the introduction. We show tall and short steps combine into decreasing diagrams, giving confluence by Theorem 1. To facilitate our presentation

we assume a rank r and a color b are given.

Under the assumption, we define *native* terms as black-rank-$(r+1)$ terms and use (non-indexed) t, s, u, v to range over them, and define *nonnative* terms as white-rank-r terms and use $\boldsymbol{t}$, $\boldsymbol{s}$, $\boldsymbol{u}$, $\boldsymbol{v}$ to range over vectors of nonnative terms, naming their individual elements by indexing and dropping the vector notation, e.g. t_1,s_n, etc.. Our choice of terminology should suggest that a term being nonnative generalises the traditional notion of the term being alien [5, p. 263], and indeed one easily checks that the aliens of a native term are nonnative, but unlike aliens, nonnative terms (and native terms as well) *are* closed under rewriting as follows from Proposition 1. This vindicates using $\rightarrow$ to denote rewriting them. By the assumption, tall native terms are top-black and tall nonnative terms top-white. Note an individual term can be native and nonnative at the same time, but the above conventions make a name unambiguously denote a term of either type: ordinary (non-indexed) names (t) denote natives, vector (boldface) names ($\boldsymbol{t}$) vectors of nonnatives, and indexed names (t_i) denote individual nonnatives.

Example 7. Assuming rank $r = 2$ and color $b = 1$, all terms $f(I, H(a))$, $f(I, G(a))$, $f(G(a), G(a))$, $G(a)$ in the peak in Example 5 are native and all but $G(a)$ are tall. The alien vector of $f(I, G(a))$ is $(I$,$G(a))$, consisting of the short nonnative I and the tall nonnative $G(a)$.

Base contexts, ranged over by C, D, E, are contexts obtained by replacing all tall aliens of some native term by the empty context $\Box$. Clearly, their rank does not exceed r.

Proposition 2. *If t is a native term, then there are a unique base context C and vector $\boldsymbol{t}$ of tall aliens,* the *base context and* base *vector of t, such that $t = C[\boldsymbol{t}]$.*

Proof. The base context is obtained by replacing all tall aliens of t by $\Box$. Uniqueness follows from uniqueness of the vector of tall aliens. $\Box$

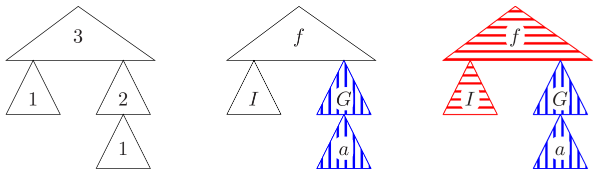


Fig. 2. From rank to base context–base vector decompositon

Example 8. For r,b as in Example 7, the base context of $f(I, G(a))$ is $f(I, \Box)$, and its base vector is $(G(a))$ (see Figure 2, base vector vertically striped, base context horizontally). The base context of $G(a)$ is the term itself, and its base vector is empty.

In order to mediate between reductions in the base context and in the vector of aliens, it will turn out convenient to have a class of terms with 'named holes in which nonnative terms can be plugged in'. For that purpose we assume to have a set of *nonnative* variables, disjoint from the term variables, and we let $\boldsymbol{x}$ range over vectors of nonnative variables. *Nonnative* substitutions are substitutions of nonnative terms for nonnative variables, ranged over by τ, σ, υ, ϕ, η. In particular, we let η denote an arbitrary but fixed bijection from nonnative variables to tall nonnative terms. A *base* term, ranged over by c, d, e, f, is a term which is either obtained by applying η^{-1} to all tall aliens of some native term, or is a nonnative variable called an *empty* base.[1] Again, their rank does not exceed r.

Proposition 3. *If t is a native term, then there is a unique non-empty base term c,* the *base of t, such that $t = c^{\eta}$.*

Proof. As for Proposition 2, also using bijectivity of η for uniqueness. ☐

The intuition that (non-empty) base terms are base contexts with 'named holes for nonnative terms' is easily seen to be correct: If C, $\boldsymbol{t}$, and c are the base context, base vector, and base of t, respectively, then $C = c^{x_i \mapsto \Box}$ and $c = C[\eta^{-1}(\boldsymbol{t})]$.

Example 9. For r,b as above, let η map x_1 to the tall nonnative $G(a)$. The base of the tall native $f(I, G(a))$ is obtained by applying η^{-1} to its tall alien $G(a)$, yielding $f(I, x_1)$. The base of $G(a)$, now seen as short native, is the term itself.

Definition 2. *A step on the nonnative vector $\boldsymbol{t}$ is* tall *if the element t_i rewritten is. The* imbalance $\#\boldsymbol{t}$ *of a vector $\boldsymbol{t}$ of nonnative terms is the cardinality of its subset of tall ones, and the* imbalance $\#t$ *of a native term $t = C[\![\boldsymbol{t}]\!]$ is $\#\boldsymbol{t}$. We write $\boldsymbol{t} \rhd\!\!\rhd_{\iota} \boldsymbol{s}$ to denote that the vector $\boldsymbol{t}$ of nonnative terms rewrites in a positive number of tall steps to $\boldsymbol{s}$ having imbalance $\iota = \#\boldsymbol{s}$, and we write $t \rhd\!\!\rhd_{\iota} s$ for a native term $t = C[\![\boldsymbol{t}]\!]$, if $\boldsymbol{t} \rhd\!\!\rhd_{\iota} \boldsymbol{s}$ and $C[\boldsymbol{s}] = s$.*

The final line is justified by the fact that then $\#\boldsymbol{s} = \iota = \#s$.

[1] Directly defining base terms inductively is not difficult but a bit cumbersome.

Example 10.
- $f(G(a),G(a)) \rhd\!\!\rhd_1 f(I,H(a))$ because the vector of tall aliens $(G(a),G(a))$ of the former rewrites by tall steps first to $(I,G(a))$ and then to $(I,H(a))$, which has imbalance 1 since it has only 1 tall element.
- $f(G(a),G(a)) \rhd\!\!\rhd_2 f(H(a),G(a))$ by rewriting the first tall alien $G(a)$ to $H(a)$. Imbalance was increased from 1 to 2 along this tall step.

Definition 3. *On nonnative vectors and native terms, a* short *step is defined to be a $\to$-step which is not of the shape of a single-step $\rhd\!\!\rhd_\iota$ rewrite sequence, and $\blacktriangleright\!\!\blacktriangleright$ denotes a positive number of short steps, with on native terms the condition that only the last step may be* tall-collapsing*, i.e. top-collapsing a tall term.*

Example 11.
- $f(G(a),G(a)) \blacktriangleright\!\!\blacktriangleright G(a)$ because the former rewrites in one step to the latter, but not by a step in a tall alien. Although also $G(a) \blacktriangleright\!\!\blacktriangleright I$ (there are no tall aliens), we do *not* have $f(G(a),G(a)) \blacktriangleright\!\!\blacktriangleright I$, since the condition that only the last step may be tall-collapsing would then be violated.[2]
- $f(I,G(a)) \to f(I,I) \to f(I,K)$ induces $f(I,G(a)) \rhd\!\!\rhd_0 f(I,I) \blacktriangleright\!\!\blacktriangleright f(I,K)$ (see Figure 3). Note that the second, short, step does not change imbalance.

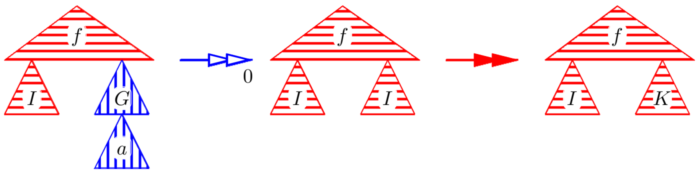


Fig. 3. Tall and short step

- Tall and short steps need not be disjoint: setting for this example rank $r = 1$ and color $b = 1$, the step $f(I,J) \to f(J,J)$ in the disjoint union of $\{f(x,y) \to f(y,y)\}$ and $\{I \to J\}$ can be both of the shape of a single-step $\rhd\!\!\rhd_\iota$ reduction and not, depending on whether I or the whole term is contracted. Hence, both $f(I,J) \rhd\!\!\rhd_1 f(J,J)$ and $f(I,J) \blacktriangleright\!\!\blacktriangleright f(J,J)$!

Proposition 4. *Confluence of $\to$ and $\blacktriangleright\!\!\blacktriangleright \cup \bigcup_\iota \rhd\!\!\rhd_\iota$ are equivalent, both on nonnative vectors and on native terms.*

Proof. In either case, it suffices to show that the reflexive–transitive closures of $\to$ and $\blacktriangleright\!\!\blacktriangleright \cup \bigcup_\iota \rhd\!\!\rhd_\iota$ coincide, which follows by monotonicity of taking reflexive–transitive closures from $\to \subseteq \blacktriangleright\!\!\blacktriangleright \cup \bigcup_\iota \rhd\!\!\rhd_\iota \subseteq \to^+$, which holds by definition. $\square$

Lemma 1. *If $t = c^\eta \blacktriangleright\!\!\blacktriangleright s$, then $s = d^\eta$ for some $c \twoheadrightarrow d$, where c is the base of t.*

[2] Transitivity of $\blacktriangleright\!\!\blacktriangleright$ on native terms could be regained by dropping the condition that only the last step may be tall-collapsing. However, this would necessitate reformulating Lemma 1 below, and would make the proof of Theorem 3 a bit more complex.

Proof. We claim that if $t = c^{\eta} \blacktriangleright\!\!\blacktriangleright s$ by a single short step with c the base of t, then $s = d^{\eta}$ for some $c \twoheadrightarrow d$ such that d is either the base of s, or it is an empty base and the step was tall-collapsing. To see this, note the redex-pattern contracted does not occur in a tall alien as the step was assumed short, and neither does it overlap a tall alien as left-hand sides of rules are *monochromatic*, i.e. either black or white. Thus, the redex-pattern is entirely contained in c, and the claim easily follows. We conclude by induction on the length of the reduction using that only its last step may be tall-collapsing. □

Example 12. For $f(G(a), G(a)) \blacktriangleright\!\!\blacktriangleright G(a)$ as in the first item of Example 10, $c = f(x_1, x_1)$ and $d = x_1$. The final part of the same item shows that without the condition that only the last step may be tall-collapsing, the result would fail: I cannot be written as d^{η} with $f(x_1, x_1) \twoheadrightarrow d$; as I is not tall, taking x_1 fails.

The following lemma is key for dealing with non-left-linear rules. Assuming confluence, it allows to construct for diverging reductions on a vector of nonnative terms, a common reduct *as balanced as* the initial vector. Intuitively, if the initial vector 'instantiates' a non-linear left-hand side, the common reduct will do so too because it is as balanced. Moreover, the imbalance of each of the converging reductions does not exceed that of the corresponding diverging one, which will be (our) key for applying the decreasing diagrams technique (Theorem 1).

Lemma 2. *If $\to$ is confluent on nonnative vectors and $\boldsymbol{t} \vartriangleright\!\!\vartriangleright_{\iota_k} \boldsymbol{s_k}$ for $1 \leq k \leq n$, then there exist $\boldsymbol{v}$ such that $\boldsymbol{t} \propto \boldsymbol{v}$ and, for $1 \leq k \leq n$, there exists $\boldsymbol{u_k}$ such that $\boldsymbol{s_k} \vartriangleright\!\!\vartriangleright^{=}_{\#\boldsymbol{v}} \boldsymbol{u_k} \blacktriangleright\!\!\blacktriangleright^{=} \boldsymbol{v}$ and $\iota_k \geq \#\boldsymbol{v}$; if moreover $n = 1$ and $\boldsymbol{s_1} \neq \boldsymbol{u_1}$, then $\iota_1 > \#\boldsymbol{v}$.*

Proof. We proceed in *stages*, which are defined to be vectors of length $n|\boldsymbol{t}|$, written using underlining in order to distinguish them from the boldfaced vectors of length $|\boldsymbol{t}|$. Starting at $\underline{s} = \boldsymbol{s_1} \dots \boldsymbol{s_n}$, each stage $\underline{v}$ will be such that $\underline{s} \twoheadrightarrow \underline{v}$ and $\underline{s} \propto \underline{v}$, until finally $\boldsymbol{t}^n \propto \underline{v}$ holds, where $\boldsymbol{t}^n$ denotes the n-fold repetition of $\boldsymbol{t}$.

By the latter condition, if the procedure stops, then $\underline{v}$ is the n-fold repetition of some vector, say $\boldsymbol{v}$. By the invariant both $\boldsymbol{s_k} \twoheadrightarrow \boldsymbol{v}$ and $\boldsymbol{s_k} \propto \boldsymbol{v}$. By the latter, we may assume that reductions of the former taking place on identical elements of $\boldsymbol{s_k}$, are in fact identical. Using that the rank of terms does not increase along rewriting, $\boldsymbol{s_k} \twoheadrightarrow \boldsymbol{v}$ has a *tall–short* factorisation[3] of shape $\boldsymbol{s_k} \vartriangleright\!\!\vartriangleright^{=}_{\#\boldsymbol{v}} \boldsymbol{u_k} \blacktriangleright\!\!\blacktriangleright^{=} \boldsymbol{v}$ with $\iota_k = \#\boldsymbol{s_k} \geq \#\boldsymbol{u_k}$ (since identical elements are reduced identically), and $\#\boldsymbol{u_k} = \#\boldsymbol{v}$ (since base reductions do not affect imbalance). This gives partial correctness of the construction. To get total correctness it must also terminate.

If stage $\underline{v}$ is not yet final, there are $\underline{t}_i = \underline{t}_j$ such that $\underline{v}_i \neq \underline{v}_j$. Per construction $\underline{t} \twoheadrightarrow \underline{s} \twoheadrightarrow \underline{v}$ hence $\underline{v}_i \twoheadleftarrow \underline{t}_i = \underline{t}_j \twoheadrightarrow \underline{v}_j$ giving some $\underline{v}_i \twoheadrightarrow w \twoheadleftarrow \underline{v}_j$ by the confluence assumption. The next stage is obtained by applying the reductions $\underline{v}_i \twoheadrightarrow w$ and $\underline{v}_j \twoheadrightarrow w$ at all indices of $\underline{v}$ to which either is applicable, i.e. to all elements identical to either $\underline{v}_i$ or $\underline{v}_j$. As the cardinality (as set) of the resulting stage is smaller than that of $\underline{v}$ (elements which were identical still are and the elements at indices i, j have become so), the procedure terminates in less than $n|\boldsymbol{t}|$ steps.

[3] W.l.o.g. we assume the factorisation to depend functionally on the given reduction.

If moreover $n = 1$, note that tall nonnative terms are only reduced when they are joined with some other term. Then note that joining, both with another tall term and with another short term (to a short term!), decreases imbalance. $\square$

Example 13. For the nonnative vector $\boldsymbol{t} = (G(a),G(a))$ with imbalance 1, consider both $\boldsymbol{t} \rhd\!\!\rhd_2 (H(a),G(a)) = \boldsymbol{s_1}$ and $\boldsymbol{t} \rhd\!\!\rhd_1 (G(a),I) = \boldsymbol{s_2}$. Then in the proof of the lemma the following vectors of nonnatives may be constructed:

$$\begin{aligned} \boldsymbol{t}^2 = \underline{t} &= (\ G(a),G(a),G(a),G(a)\ ) \\ \boldsymbol{s_1 s_2} = \underline{s} &= (\ H(a),G(a),G(a),\ \ I\ \ \ ) \\ \underline{u} &= (\ H(a),H(a),H(a),\ \ I\ \ \ ) \\ \boldsymbol{v}^2 = \underline{v} &= (\ \ \ K\ \ ,\ \ K\ \ ,\ \ K\ \ ,\ \ K\ \ \ ) \end{aligned}$$

In stage $\underline{s}$, $\underline{s}_1 = H(a) \neq G(a) = \underline{s}_2$, but $\underline{t}_1 = \underline{t}_2$. By confluence, the latter are joinable, e.g., by $H(a) \leftarrow G(a)$, and applying this to all terms in $\underline{s}$ yields $\underline{u}$.

In stage $\underline{u}$, $\underline{s}_1 = H(a) \neq I = \underline{s}_4$, but $\underline{t}_1 = \underline{t}_4$. By confluence, the latter are joinable, e.g., by $H(a) \twoheadrightarrow K \leftarrow I$, and applying these to all terms in $\underline{u}$ yields $\underline{v}$.

Therefore, for $1 \leq k \leq 2$, $\boldsymbol{s_k} \twoheadrightarrow (K,K) = \boldsymbol{v}$, with tall–short factorisations $\boldsymbol{s_1} \rhd\!\!\rhd_0 (J,J) \blacktriangleright\!\!\blacktriangleright \boldsymbol{v}$ and $\boldsymbol{s_2} \rhd\!\!\rhd_0 (J,I) \blacktriangleright\!\!\blacktriangleright \boldsymbol{v}$, both with imbalance (0) not exceeding that of $\boldsymbol{t}$ (1).

Lemma 3. *1. If $\boldsymbol{t} \rhd\!\!\rhd_\iota \boldsymbol{s} \blacktriangleright\!\!\blacktriangleright \boldsymbol{u}$, then $C[\boldsymbol{t}] \rhd\!\!\rhd_\iota C[\boldsymbol{s}] \blacktriangleright\!\!\blacktriangleright C[\boldsymbol{u}]$, for base contexts C.*
2. If $c \twoheadrightarrow d$, then for any nonnative substitution τ, $c^\tau \blacktriangleright\!\!\blacktriangleright^= d^\tau$ and $\#c^\tau \geq \#d^\tau$.

Proof. The items follow from closure of rewriting under contexts and substitutions; $\#c^\tau \geq \#d^\tau$ since reducing c can only replicate the tall nonnatives in τ. $\square$

Example 14. 1. Filling the base context $f(\square,\square)$ with the tall–short factorisations of $\boldsymbol{s_k} \twoheadrightarrow \boldsymbol{v}$ in Example 13, yields $f(H(a),G(a)) \rhd\!\!\rhd_0 f(J,J) \blacktriangleright\!\!\blacktriangleright f(K,K)$ and $f(G(a),I) \rhd\!\!\rhd_0 f(J,I) \blacktriangleright\!\!\blacktriangleright f(K,K)$.
2. Since $f(x_1,x_1) \to x_1$, we have $f(G(a),G(a)) \blacktriangleright\!\!\blacktriangleright G(a)$ by mapping x_1 to $G(a)$, but also $f(a,a) \blacktriangleright\!\!\blacktriangleright a$ by mapping x_1 to a.

The proof of our main theorem is by induction on the rank of a peak. The idea is to decompose a peak into tall and short peaks, which being of lower rank can both be completed into valleys by the induction hypothesis, which then can be combined to yield decreasing diagrams by measuring tall reductions by their imbalance, giving confluence by the decreasing diagrams theorem.

Theorem 3. *$\to$ is confluent if and only if both $\to_1$ and $\to_2$ are.*

Proof. The direction from left to right is simple: For any color c, a $\to_c$-peak 'is' a $\to$-peak of c-rank-1 terms. By Con($\to$) and Proposition 1, the latter peak can be completed by a $\to$-valley of c-rank-1 terms, which 'is' a $\to_c$-valley.

The direction from right to left is complex. Assuming Con($\to_1$) and Con($\to_2$), we prove by induction on the rank, that any $\to$-peak of terms of at most that rank can be completed by a $\to$-valley of such terms again.

The base case, a peak of terms of rank 1, is simple and symmetrical to the above: such a peak 'is' a $\to_c$-peak for some color c. By $\mathrm{Con}(\to_c)$ it can be completed by a $\to_c$-valley, which 'is' a $\to$-valley of c-rank-1 terms.

Using the terminology introduced above, the step case corresponds to proving confluence of reduction on native (black-rank-$(r+1)$) terms, having confluence of reduction on nonnative (white-rank-r) terms as induction hypothesis. By Proposition 4 it suffices to show confluence of $\mathrel{\blacktriangleright\!\!\blacktriangleright} \cup \bigcup_\iota \mathrel{\rhd\!\!\rhd}_\iota$. To that end, we show the preconditions of Theorem 1 are satisfied taking the $\mathrel{\rhd\!\!\rhd}_\iota$ and $\mathrel{\blacktriangleright\!\!\blacktriangleright}$ as *steps*, ordering $\mathrel{\rhd\!\!\rhd}_\iota$ above $\mathrel{\rhd\!\!\rhd}_\kappa$ if $\iota > \kappa$, and ordering all these above $\mathrel{\blacktriangleright\!\!\blacktriangleright}$.

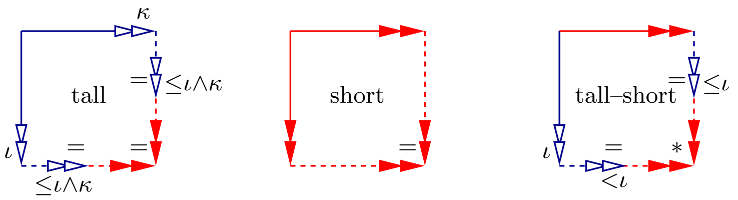


Fig. 4. Decreasing diagrams case analysis

We distinguish cases (see Figure 4) on the types, short or tall, of the steps in a peak having the native term t as source, and we let C, $\boldsymbol{t}$, and c be the base context, base vector, and base of t as given by Propositions 2 and 3, respectively.

(tall–tall) Suppose $s \mathrel{\lhd\!\!\lhd}_\iota t \mathrel{\rhd\!\!\rhd}_\kappa u$. By definition $s = C[\boldsymbol{s}] \mathrel{\lhd\!\!\lhd}_\iota C[\boldsymbol{t}] \mathrel{\rhd\!\!\rhd}_\kappa C[\boldsymbol{u}] = u$ for some $\boldsymbol{s} \mathrel{\lhd\!\!\lhd}_\iota \boldsymbol{t} \mathrel{\rhd\!\!\rhd}_\kappa \boldsymbol{u}$. By the induction hypothesis we may apply Lemma 2 giving $\boldsymbol{s} \mathrel{\rhd\!\!\rhd}^=_{\#\boldsymbol{v}} \boldsymbol{s'} \mathrel{\blacktriangleright\!\!\blacktriangleright}^= \boldsymbol{v} \mathrel{\blacktriangleleft\!\!\blacktriangleleft} \boldsymbol{u'} \mathrel{\lhd\!\!\lhd}_{\#\boldsymbol{v}} \boldsymbol{u}$ for some $\boldsymbol{s'}, \boldsymbol{v}, \boldsymbol{u'}$ with $\iota \ge \#\boldsymbol{v} \le \kappa$. We conclude to $s = C[\boldsymbol{s}] \mathrel{\rhd\!\!\rhd}^=_{\#\boldsymbol{v}} C[\boldsymbol{s'}] \mathrel{\blacktriangleright\!\!\blacktriangleright}^= C[\boldsymbol{v}] \mathrel{\blacktriangleleft\!\!\blacktriangleleft}^= C[\boldsymbol{u'}] \mathrel{\lhd\!\!\lhd}^=_{\#\boldsymbol{v}} C[\boldsymbol{u}] = u$ from Lemma 3(1), giving a decreasing diagram (Figure 4 left).

(short–short) Suppose $s \mathrel{\blacktriangleleft\!\!\blacktriangleleft} t \mathrel{\blacktriangleright\!\!\blacktriangleright} u$. Lemma 1 entails $s = d^\eta \mathrel{\blacktriangleleft\!\!\blacktriangleleft} c^\eta \mathrel{\blacktriangleright\!\!\blacktriangleright} e^\eta = u$ for some nonnative peak $d \twoheadleftarrow c \twoheadrightarrow e$. By the induction hypothesis for this peak, $d \twoheadrightarrow f \twoheadleftarrow e$ for some f. We conclude to $s = d^\eta \mathrel{\blacktriangleright\!\!\blacktriangleright}^= f^\eta \mathrel{\blacktriangleleft\!\!\blacktriangleleft}^= e^\eta = u$ by Lemma 3(2), giving a decreasing diagram (Figure 4 middle).

(tall–short) Suppose $s \mathrel{\lhd\!\!\lhd}_\iota t \mathrel{\blacktriangleright\!\!\blacktriangleright} u$. By definition and Lemma 1 we have $s = C[\boldsymbol{s}] \mathrel{\lhd\!\!\lhd}_\iota C[\boldsymbol{t}] = t = c^\eta \mathrel{\blacktriangleright\!\!\blacktriangleright} e^\eta = u$ for some $\boldsymbol{s} \mathrel{\lhd\!\!\lhd}_\iota \boldsymbol{t}$ and $c \twoheadrightarrow e$.
By the induction hypothesis and Lemma 2, $\boldsymbol{s} \mathrel{\lhd\!\!\lhd}_\iota \boldsymbol{t}$ entails $\boldsymbol{s} \mathrel{\rhd\!\!\rhd}^=_{\#\boldsymbol{v}} \boldsymbol{s'} \mathrel{\blacktriangleright\!\!\blacktriangleright}^= \boldsymbol{v}$ for some $\boldsymbol{s'}, \boldsymbol{v}$ with $\boldsymbol{t} \propto \boldsymbol{v}$ and $\iota \ge \#\boldsymbol{v}$, and if $\boldsymbol{s} \ne \boldsymbol{s'}$ then $\iota > \#\boldsymbol{v}$. Lemma 3(1) yields $C[\boldsymbol{s}] \mathrel{\rhd\!\!\rhd}^=_{\#\boldsymbol{v}} C[\boldsymbol{s'}] \mathrel{\blacktriangleright\!\!\blacktriangleright}^= C[\boldsymbol{v}]$ and $\boldsymbol{t} \propto \boldsymbol{v}$ gives $C[\boldsymbol{v}] = c^\sigma$ for some $\sigma \twoheadleftarrow \eta$. By Lemma 3(2), $c \twoheadrightarrow e$ entails $c^\sigma \mathrel{\blacktriangleright\!\!\blacktriangleright}^= e^\sigma$ with $\#c^\sigma \ge \#e^\sigma$. To conclude (as in Figure 4 right) we distinguish cases on whether e is empty or not.

(empty) If e is empty, say $e = x_i$ then $\sigma \twoheadleftarrow \eta$ entails $e^\sigma = \sigma(x_i) \mathrel{\blacktriangleleft\!\!\blacktriangleleft}^= \eta(x_i) = e^\eta$, thus $s = C[\boldsymbol{s}] \mathrel{\rhd\!\!\rhd}^=_{\#\boldsymbol{v}} C[\boldsymbol{s'}] \mathrel{\blacktriangleright\!\!\blacktriangleright}^= C[\boldsymbol{v}] = c^\sigma \mathrel{\blacktriangleright\!\!\blacktriangleright}^= e^\sigma \mathrel{\blacktriangleleft\!\!\blacktriangleleft}^= e^\eta = u$, giving a decreasing diagram since $\iota > \#\boldsymbol{v}$ if the $\mathrel{\rhd\!\!\rhd}^=_{\#\boldsymbol{v}}$-step is non-empty.

(non-empty) If e is not empty, let E and $\boldsymbol{x}$ be the (unique) base context and vector of nonnative variables such that $E[\boldsymbol{x}] = e$. Then $\sigma \twoheadleftarrow \eta$

entails $\boldsymbol{x}^\sigma \twoheadleftarrow \boldsymbol{x}^\eta$, any tall–short factorisation (see Lemma 2) of which yields $e^\sigma = E[\boldsymbol{x}^\sigma] \twoheadleftarrow^{=} E[\boldsymbol{x}^\phi] \lhd\!\!\lhd^{=}_{\kappa} E[\boldsymbol{x}^\eta] = e^\eta$ for some ϕ by Lemma 3(1), thus $s = C[\boldsymbol{s}] \rhd\!\!\rhd^{=}_{\#\boldsymbol{v}} C[\boldsymbol{s'}] \twoheadrightarrow^{=} C[\boldsymbol{v}] = c^\sigma \twoheadrightarrow^{=} e^\sigma \twoheadleftarrow^{=} E[\boldsymbol{x}^\phi] \lhd\!\!\lhd^{=}_{\kappa} e^\eta = u$, giving a decreasing diagram since $\iota > \#\boldsymbol{v}$ if the $\rhd\!\!\rhd^{=}_{\#\boldsymbol{v}}$-step is non-empty as before, and $\iota \geq \#\boldsymbol{v} = \#C[\boldsymbol{v}] = \#c^\sigma \geq \#e^\sigma = \#E[\boldsymbol{x}^\phi] = \kappa$. $\square$

$$\begin{tikzcd}
f(G(a),G(a)) \arrow[rrrr, twoheadrightarrow] \arrow[d, "1"] & & & & G(a) \arrow[d, twoheadrightarrow] \\
f(I,G(a)) \arrow[rr, dashed, "0"'] \arrow[d, "1"] & & f(I,I) \arrow[rr, dashed, twoheadrightarrow] \arrow[d, twoheadrightarrow] \arrow[u, phantom, "\text{tall--short}"] & & I \arrow[d, twoheadrightarrow] \\
f(I,H(a)) \arrow[r, dashed, "0"'] & f(I,J) \arrow[r, dashed, twoheadrightarrow] \arrow[u, phantom, "\text{tall}"] & f(I,K) \arrow[r, dashed, twoheadrightarrow] & f(K,K) \arrow[r, dashed, twoheadrightarrow] \arrow[u, phantom, "\text{short}"] & K
\end{tikzcd}$$

Fig. 5. The three cases in Theorem 3

Example 15. Successively completing the peaks of Example 5, written as:

$$f(I,H(a)) \lhd\!\!\lhd_1 f(I,G(a)) \lhd\!\!\lhd_1 f(G(a),G(a)) \twoheadrightarrow G(a)$$

into valleys, gives rise to Figure 5, illustrating the three cases in Theorem 3.

The main difference between our proof of modularity of confluence and those in the literature [1,2,3] is the way in which they deal with the *identity-check* of the terms occurring at the non-linear argument places of the left-hand side of a rule. Reducing these terms may cause that the identity-check ceases to succeed, upon which so-called *balancing* reductions must be performed on these terms, in order to make the identity-check succeed again.

Our proof relies on the *local* reduction history, i.e. on the reductions performed since the moment the identity-check *did* succeed, to construct the balancing reductions using Lemma 2. In the above example, in order to apply the rule $f(x,x) \to x$ to the term $f(I,G(a))$, the reduction from $f(G(a),G(a))$ for which the identity-check *did* succeed, is used to construct the balancing reduction to $f(I,I)$ for which the identity-check succeeds again.

In contrast, the proofs in the literature rely on the *global* reduction history by mapping convertible terms (so certainly those terms for which the identity-check once did succeed) to some witness. In order to guarantee that all the convertible terms in fact *reduce* to the witness, i.e. in order to obtain balancing *reductions*, the witness is chosen to be of minimal rank, cf. Example 3, which has the side-effect of making the proofs non-constructive.

4 Modularity of Confluence, Constructed

A proof is *constructive* if it demonstrates the existence of a mathematical object by providing a procedure for creating it. A rewriting system with a constructive proof of confluence is *constructively* confluent. More precisely, we require to have a procedure which for any given peak, say given as proof terms in rewriting logic [4, Chapter 8], constructs the corresponding valley. In other words, we require the existence of effective confluence functions f, g as in Figure 6.

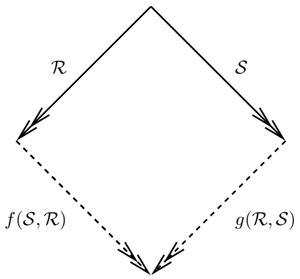


Fig. 6. Effective confluence functions

Example 16. The term rewriting system $\mathcal{CL}$ in the introduction is constructively confluent, which can be seen by setting both f and g to the residual/projection function, which is effective [4, Definition 8.7.54] (giving rise to least upper bounds [4, Figure 8.7.52]), because $\mathcal{CL}$ is an orthogonal TRS.

The term rewriting system $\mathcal{E}$ is constructively confluent, which can be seen by setting both f and g to the binary function computing the normal form of (the target of) its second argument. This procedure is effective for any constructively terminating TRS, such as $\mathcal{E}$, the critical pairs of which are convergent (the upper bounds computed are greatest with respect to the induced partial order $\twoheadrightarrow$).

Corollary 1. *Constructive confluence is modular.*

Proof. All constructions in Section 3 are effective. In fact, the proof by induction of Theorem 3 gives rise to a program which in order to compute a common reduct for a peak of a given rank, relies on the ability to decompose the peak into peaks of lower rank, on making recursive calls to itself giving valleys for those lower ranks, and the ability to compose these valleys again.

That the program eventually terminates, i.e. that it does not produce an infinitely regressing sequence of peaks for which common reducts need to be found, relies on Theorem 1 the proof of which is seen to be constructive.[4] □

Example 17. The disjoint union of $\mathcal{CL} \uplus \mathcal{E}$ is constructively confluent, computing least (greatest) upper bounds on $\mathcal{CL}$ ($\mathcal{E}$) components.

Note that since the confluence proofs in the literature are not constructive, they yield no method better than a blind search to find a common reduct, the

[4] Although, as far as we know, Theorem 1 itself has not been formalized in some proof checker, its 'point version' (where objects instead of steps are labelled) has [7].

termination of which will not be constructive. Even if one is not concerned with that, such a blind search is of course extremely inefficient.

Remark 1. The complexity of our construction should be studied. It should be interesting to formalize our proof in, say Coq, and extract an effective confluence function which computes for any pair of diverging reductions in a disjoint union, the corresponding converging reductions, by making calls to the confluence functions for the component term rewriting systems.

5 (Im)possible Extensions

5.1 Constructor-Sharing TRSs

Two (possibly non-disjoint) TRSs $\mathcal{T}_1$ and $\mathcal{T}_2$ are said to be *constructor-sharing* if the symbols shared by their alphabets (possibly none) are *constructors*, i.e. do not appear as head-symbols of left-hand sides of rules. The following well-known example [5, Examples 8.2.1,8.5.24] shows that the union of confluent constructor-sharing TRSs need not be confluent, in general.

Example 18. Let $\mathcal{T}_1$ be $\{\infty \to S(\infty)\}$ and $\mathcal{T}_2$ be $\{E(x,x) \to \mathit{true}, E(x,S(x)) \to \mathit{false}\}$. Then $\mathcal{T}_1$ and $\mathcal{T}_2$ are confluent, and share the constructor S, but their union $\mathcal{T}_1 \cup \mathcal{T}_2$[5] is not confluent: $\mathit{true} \leftarrow E(\infty,\infty) \to E(\infty,S(\infty)) \to \mathit{false}$. A similar counterexample arises when setting $\mathcal{T}_1$ to $\{\infty \to I(S(\infty)), I(x) \to x\}$.

The counterexamples can be barred by forbidding the TRSs to have rules with a shared constructor or variable as head-symbol of the right-hand side, forbidding $\infty \to S(\infty)$ and $I(x) \to x$ in the example. Modularity of (constructive) confluence of such *opaque* TRSs [5, Corollary 8.5.23] reduces to ordinary modularity:

Definition 4. *For $\mathcal{T} = (\Sigma, R)$ an opaque TRS, $\underline{\mathcal{T}}$ is $(\underline{\Sigma}, \underline{R})$ with:*

- *$\underline{\Sigma}$ a new signature having opaque Σ-contexts as function symbols, where a Σ-context is* opaque *if only its head-symbol is not a constructor; its arity as function symbol is the number of holes in it as context.*
- *For a Σ-term t not headed by a shared constructor, let $\underline{t}$ denote the (unique) corresponding $\underline{\Sigma}$-term. Then $\underline{R}$ is a new set of rules consisting of $\underline{l^\gamma} \to \underline{r^\gamma}$ for all $l \to r \in R$ and all substitutions γ of shared constructor terms.*

Example 19. For the opaque TRS $\mathcal{T}_2$ of Example 18, $\underline{\mathcal{T}}_2$ consists of *false*, *true*, $E(\Box,\Box)$, $E(S(\Box),\Box)$, $E(\Box,S(\Box))$, $E(S(\Box),S(\Box))$, $E(S(S(\Box)),\Box)$,...

Lemma 4. *For any opaque TRS $\mathcal{T}$, $Con(\mathcal{T}) \iff Con(\underline{\mathcal{T}})$.*

Proof (Sketch). Since shared constructor symbols are inert, confluence of $\mathcal{T}$ reduces to confluence of non-shared-constructor-headed terms. We conclude as by opaqueness $\underline{\cdot}$ is a bisimulation [4, Section 8.4.1] between $\to_{\mathcal{T}}$ and $\to_{\underline{\mathcal{T}}}$ on such terms.

[5] Huet's counterexample to confluence of non-overlapping TRS [4, Example 4.1.4(i)].

Theorem 4. *(Constructive) confluence is modular for opaque TRSs.*

Proof. Suppose $\mathcal{T}_1$ and $\mathcal{T}_2$ are confluent opaque TRSs. By the lemma and Theorem 3: $\mathrm{Con}(\mathcal{T}_1 \cup \mathcal{T}_2) \iff \mathrm{Con}(\underline{\mathcal{T}_1 \cup \mathcal{T}_2}) \iff \mathrm{Con}(\underline{\mathcal{T}}_1 \cup \underline{\mathcal{T}}_2) \iff \mathrm{Con}(\underline{\mathcal{T}}_1 \uplus \underline{\mathcal{T}}_2) \iff \mathrm{Con}(\underline{\mathcal{T}}_1)\ \&\ \mathrm{Con}(\underline{\mathcal{T}}_2) \iff \mathrm{Con}(\mathcal{T}_1)\ \&\ \mathrm{Con}(\mathcal{T}_2)$. □

5.2 Extra-Variable TRSs

Definition 5. *An* extra-variable *TRS is a term rewriting system where the condition that variables of the right-hand side must be contained in those of the left-hand side, is dropped.*

Modularity of confluence for extra-variable TRSs was studied in [8] in an abstract categorical setting. However, the following surprising example shows that confluence for extra-variable TRSs is *not* preserved under *de*composition, hence that the conditions on the categorical setting in [8] must exclude a rather large class of extra-variable TRSs, in order not to contradict the statement in [9, Section 4.3] that in that setting confluence *is* preserved under decomposition.

Example 20. Consider the following extra-variable TRSs, where the variable z in the right-hand side of the first rule of $\mathcal{T}_1$ is not contained in the variables of its left-hand side:

$$\mathcal{T}_1 = \{f(x,y) \to f(z,z), f(b,c) \to a, b \to d, c \to d\}$$

$$\mathcal{T}_2 = \{M(y,x,x) \to y, M(x,x,y) \to y\}$$

Clearly $\mathcal{T}_1$ is not confluent since $a \leftarrow f(b,c) \to f(t,t) \not\twoheadrightarrow a$ as no term t reduces to both b and c. However the disjoint union of $\mathcal{T}_1$ and $\mathcal{T}_2$ *is* confluent, as then

$$f(x,y) \to f(M(b,d,c), M(b,d,c)) \to^2 f(M(b,d,d), M(d,d,c)) \to^2 f(b,c) \to a$$

Formally: this justifies adjoining the rule $\varrho : f(x,y) \to a$, after which the first two rules of $\mathcal{T}_1$ are derivable: $f(x,y) \to_\varrho a \leftarrow_\varrho f(z,z)$ and $f(b,c) \to_\varrho a$. Removing these two rules we obtain an ordinary TRS $\mathcal{T}$ confluence of which entails confluence of the disjoint union as per construction $\leftrightarrow^*_{\mathcal{T}_1 \uplus \mathcal{T}_2} = \leftrightarrow^*_{\mathcal{T}}$ and $\twoheadrightarrow_{\mathcal{T}} \subseteq \twoheadrightarrow_{\mathcal{T}_1 \uplus \mathcal{T}_2}$. Confluence of $\mathcal{T}$ holds by Huet's Critical Pair Lemma [4, Thm. 2.7.15], as $\mathcal{T}$ is terminating and its critical pair, arising from $x \leftarrow M(x,x,x) \to x$, is trivial.

Remark 2. Note that in the above counterexample, the disjoint union $\mathcal{T}_1 \uplus \mathcal{T}_2$ is neither orthogonal nor left- or right-linear. We conjecture that requiring either of these three properties suffices for establishing that the components of a confluent extra-variable TRS are confluent again.

At present, we do not know whether our method does extend to show the other direction, preservation of confluence under disjoint unions, for extra-variable TRSs. We conjecture it does, but the following two examples exhibit some proof-invariant-breaking phenomena.

Example 21. It is not always possible to avoid creating aliens to find a common reduct. For instance, the peak $H(a) \leftarrow f(H(a)) \rightarrow g(H(a), a)$ in the union of

$$\mathcal{T}_1 = \{f(x) \rightarrow x, f(x) \rightarrow g(x, a), g(x, y) \rightarrow g(x, z), g(x, x) \rightarrow x\}$$

with the empty TRS $\mathcal{T}_2$ over $\{H\}$, can only be completed into a valley of shape $H(a) \leftarrow g(H(a), H(a)) \twoheadleftarrow g(H(a), a)$, i.e. by creating an alien $H(a)$.

Example 22. It is not always possible to erase an alien once created. For instance, consider extending the rules expressing that $*$ is an associative–commutative symbol with the creation *ex nihilo* rule $x * y \rightarrow x * y * z$. This gives a confluent TRS, where finding a common reduct can be thought of as constructing a multiset union. Obviously, reduction in the disjoint union with an arbitrary other (non-erasing) TRS may create terms of arbitrary rank which can never be erased.

Remark 3. According to [10], the proof of [3] that confluence is preserved does extend to extra-variable TRSs.

5.3 Pseudo-TRSs

For *pseudo*-TRSs [4, p. 36], i.e. extra-variable TRSs where also the condition that the head-symbol of the left-hand side must be a function symbol, is dropped, i.e. TRSs where both the left- and right-hand side are arbitrary terms, modularity trivially fails in both directions:

- The disjoint union of the confluent pseudo-TRSs $\mathcal{T}_1 = \{x \rightarrow f(x)\}$ and $\mathcal{T}_2 = \{G(A) \rightarrow A\}$ is not confluent. To wit $A \leftarrow G(A) \rightarrow G(f(A))$, and A, $G(f(A))$ do not have a common reduct as one easily verifies.
- Vice versa, the disjoint union of the pseudo-TRSs $\mathcal{T}_1 = \{a \rightarrow b, a \rightarrow c\}$ and $\mathcal{T}_2 = \{x \rightarrow D\}$ is confluent since any term reduces in one step to D, but clearly $\mathcal{T}_1$ is not confluent.

6 Conclusion

We have presented a novel proof of modularity of (constructive) confluence for term rewriting systems.

- Our proof is relatively short, in any case when omitting the illustrative examples from Section 3. Still a better measure for that would be obtained by formalising it, which we intend to pursue.
- Our proof is itself modular in that it is based on the decreasing diagrams technique. Following [11], it would be interesting to pursue this further and try to factorize other existing proof methods in this way as well. For instance, the commutation of outer and inner reductions for preserved terms ([4, Lemma 5.8.7(ii)],[5, Lemma 8.5.15]) can be proven by labelling inner steps by the imbalance of their target, since that gives rise to the *same* decreasing diagrams as in Figure 4, after replacing short (tall) by inner (outer).[6]

[6] Still, that wouldn't make these proofs constructive, as the projection to preserved terms they rely on is not constructive.

– It would be interesting to see whether our novel way of decomposing terms and steps into short and tall parts, could be meaningfully employed to establish other results (constructively), e.g. persistence of confluence [12], modularity of uniqueness of normal forms [13], or modularity of confluence for conditional TRSs [14].

Acknowledgments. We thank the anonymous referees, Bernhard Gramlich, Jean-Pierre Jouannaud, Jeroen Ketema, Paul-André Melliès, Aart Middeldorp, and attendants of the TeReSe seminar (Utrecht, 2006) and the Austria–Japan Summer Workshop (Obergurgl, 2007) for feedback.

References

1. Toyama, Y.: On the Church–Rosser property for the direct sum of term rewriting systems. Journal of the Association for Computing Machinery 34(1), 128–143 (1987)
2. Klop, J., Middeldorp, A., Toyama, Y., de Vrijer, R.: Modularity of confluence: A simplified proof. Information Processing Letters 49(2), 101–109 (1994)
3. Jouannaud, J.: Modular Church–Rosser modulo. In: Pfenning, F. (ed.) RTA 2006. LNCS, vol. 4098, pp. 96–107. Springer, Heidelberg (2006)
4. Terese: Term Rewriting Systems. Cambridge Tracts in Theoretical Computer Science, vol. 55. Cambridge University Press, Cambridge (2003)
5. Ohlebusch, E.: Advanced Topics in Term Rewriting. Springer, Heidelberg (2002)
6. van Oostrom, V.: Confluence by decreasing diagrams. Theoretical Computer Science 126(2), 259–280 (1994)
7. Bognar, M.: A point version of decreasing diagrams. In: Engelfriet, J., Spaan, T. (eds.) Proceedings Accolade 1996, pp. 1–14. Amsterdam (1997); Dutch Graduate School in Logic
8. Lüth, C.: Compositional term rewriting: An algebraic proof of Toyama's theorem. In: Ganzinger, H. (ed.) RTA 1996. LNCS, vol. 1103, pp. 261–275. Springer, Heidelberg (1996)
9. Lüth, C.: Categorical Term Rewriting: Monads and Modularity. PhD thesis, University of Edinburgh (1997)
10. Jouannaud, J.: Personal communication (2006)
11. van Oostrom, V.: Confluence by decreasing diagrams, converted. In: Voronkov, A. (ed.) Proceedings of the 19th RTA. LNCS, vol. 5117, pp. 306–320. Springer, Heidelberg (2008)
12. Aoto, T., Toyama, Y.: Persistency of confluence. Journal of Universal Computer Science 3, 1134–1147 (1997)
13. Middeldorp, A.: Modular aspects of properties of term rewriting systems related to normal forms. In: Dershowitz, N. (ed.) RTA 1989. LNCS, vol. 355, pp. 263–277. Springer, Heidelberg (1989)
14. Middeldorp, A.: Confluence of the disjoint union of conditional term rewriting systems. In: Okada, M., Kaplan, S. (eds.) CTRS 1990. LNCS, vol. 516, pp. 295–306. Springer, Heidelberg (1991)

Automated Complexity Analysis Based on 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. In this paper, we present a variant of the dependency pair method for analysing runtime complexities of term rewrite systems automatically. This method is easy to implement, but significantly extends the analytic power of existing direct methods. Our findings extend the class of TRSs whose linear or quadratic runtime complexity can be detected automatically. 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. In order to assess the complexity of a (terminating) term rewrite system (TRS for short) it is natural to look at the maximal length of derivation sequences, as suggested by Hofbauer and Lautemann in [1]. More precisely, the *derivational complexity function* with respect to a (terminating and finitely-branching) TRS $\mathcal{R}$ relates the length of the longest derivation sequence to the size of the initial term. For direct termination techniques it is often possible to establish upper-bounds on the growth rate of the derivational complexity function from the termination proof of $\mathcal{R}$, see for example [1,2,3,4,5,6].

However, if one is interested in methods that induce feasible (i.e., polynomial) complexity, the existing body of research is not directly applicable. On one hand this is due to the fact that for standard techniques the derivational complexity cannot be contained by polynomial growth rates. (See [6] for the exception to the rule.) Already termination proofs by polynomial interpretations induce a double-exponential upper-bound on the derivational complexity, cf. [1]. On the other hand this is—to some extent—the consequence of the *definition* of derivational complexity as this measure does not discriminate between different types of initial terms, while in modelling declarative programs the type of the initial term is usually quite restrictive. The following example clarifies the situation.

* This research is partly supported by FWF (Austrian Science Fund) project P20133, Leading Project e-Society (MEXT of Japan), and STARC.

Example 1. Consider the TRS $\mathcal{R}$

$$\begin{array}{llll} 1\colon & x - \mathsf{0} \to x & 3\colon & \mathsf{0} \div \mathsf{s}(y) \to \mathsf{0} \\ 2\colon & \mathsf{s}(x) - \mathsf{s}(y) \to x - y & 4\colon & \mathsf{s}(x) \div \mathsf{s}(y) \to \mathsf{s}((x - y) \div \mathsf{s}(y)) \end{array}$$

Although the functions *computed* by $\mathcal{R}$ are obviously feasible this is not reflected in the derivational complexity of $\mathcal{R}$. Consider rule 4, which we abbreviate as $C[y] \to D[y, y]$. Since the maximal derivation length starting with $C^n[y]$ equals 2^{n-1} for all $n > 0$, $\mathcal{R}$ admits (at least) exponential derivational complexity.

After a moment one sees that this behaviour is forced upon us, as the TRS $\mathcal{R}$ may duplicate variables, i.e., $\mathcal{R}$ is *duplicating*. Furthermore, in general the applicability of the above results is typically limited to simple termination. (But see [4,5,6] for exceptions to this rule.) To overcome the first mentioned restriction we propose to study *runtime complexities* of rewrite systems. The *runtime complexity function* with respect to a TRS $\mathcal{R}$ relates the length of the longest derivation sequence to the size of the arguments of the initial term, where the arguments are supposed to be in normal form. In order to overcome the second restriction, we base our study on a fresh analysis of the *dependency pair method*. The dependency pair method [7] is a powerful (and easily automatable) method for proving termination of term rewrite systems. In contrast to the above cited direct termination methods, this technique is a *transformation* technique, allowing for applicability beyond simple termination.

Studying (runtime) complexities induced by the dependency pair method is challenging. Below we give an (easy) example showing that the direct translations of original theorems formulated in the context of termination analysis is destined to failure in the context of runtime complexity analysis. If one recalls that the dependency pair method is based on the observation that from an arbitrary non-terminating term one can extract a minimal non-terminating subterm, this is not surprising. Through a very careful investigation of the original formulation of the dependency pair method (see [7,8], but also [9]), we establish a runtime complexity analysis based on the dependency pair method. In doing so, we introduce *weak dependency pairs* and *weak innermost dependency pairs* as a general adaption of dependency pairs to (innermost) runtime complexity analysis. Here the *innermost* runtime complexity function with respect to a TRS $\mathcal{R}$ relates the length of the longest innermost derivation sequence to the size of the arguments of the initial term, where again the arguments are supposed to be in normal form.

Our main result shows how natural improvements of the dependency pair method, like *usable rules*, *reduction pairs*, and *argument filterings* become applicable in this context. Moreover, for innermost rewriting, we establish an easy criteria to decide when weak innermost dependency pairs can be replaced by "standard" dependency pairs without introducing fallacies. Thus we establish (for the first time) a method to analyse the derivation length induced by the (standard) dependency pair method for innermost rewriting. We have implemented the technique and experimental evidence shows that the use of weak dependency pairs significantly increases the applicability of the body of existing results on the estimation of derivation length via termination techniques. In particular, our findings extend the class of TRSs whose linear or quadratic runtime complexity can be detected automatically.

The remainder of this paper is organised as follows. In the next section we recall basic notions and starting points of this paper. Section 3 and 4 introduce weak dependency pairs and discuss the employability of the usable rule criteria. In Section 5 we show how to estimate runtime complexities through relative rewriting and in Section 6 we state our Main Theorem. The presented technique has been implemented and we provide ample numerical data for assessing the viability of the method. This evidence can be found in Section 7. Finally in Section 8 we conclude and mention possible future work.

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 $\mathcal{T}(\mathcal{F},\mathcal{V})$. 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 position* $\mathcal{P}\mathsf{os}(t)$ 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 subterm relation is denoted as $\trianglelefteq$. $\mathcal{V}\mathsf{ar}(t)$ denotes the set of variables occurring in a term t and the *size* $|t|$ of a term is defined as the number of symbols in t.

A *term rewrite system* (*TRS* for short) $\mathcal{R}$ 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 $\to_{\mathcal{R}}$. The transitive closure of $\to_{\mathcal{R}}$ 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 $s \to t$. With $\mathcal{NF}(\mathcal{R})$ we denote the set of all normal forms of a term rewrite system $\mathcal{R}$. The *innermost rewrite relation* $\xrightarrow{\mathrm{i}}_{\mathcal{R}}$ of a TRS $\mathcal{R}$ is defined on terms as follows: $s \xrightarrow{\mathrm{i}}_{\mathcal{R}} t$ if there exist 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 function 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 $1 \leqslant i \leqslant n$. A TRS $\mathcal{R}$ is called *duplicating* if there exists a rule $l \to r \in \mathcal{R}$ such that a variable occurs more often in r than in l. We call a TRS *terminating* if no infinite rewrite sequence exists. Let s and t be terms. If exactly n steps are performed to rewrite s to t we write $s \to^n t$. 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 $\rightarrow_{\mathcal{R}}$ ($\xrightarrow{i}_{\mathcal{R}}$) on the set $\mathcal{T}_{\mathsf{b}}$ of all *basic* terms.[1] More precisely, the *runtime complexity function* (with respect to $\mathcal{R}$) is defined as $\mathsf{rc}_{\mathcal{R}}(n) := \mathsf{rc}(n, \mathcal{T}_{\mathsf{b}}, \rightarrow_{\mathcal{R}})$ and we define the *innermost runtime complexity function* as $\mathsf{rc}^{\mathsf{i}}_{\mathcal{R}}(n) := \mathsf{rc}(n, \mathcal{T}_{\mathsf{b}}, \xrightarrow{i}_{\mathcal{R}})$. Finally, the *derivational complexity function* (with respect to $\mathcal{R}$) becomes definable as follows: $\mathsf{dc}_{\mathcal{R}}(n) = \mathsf{rc}(n, \mathcal{T}, \rightarrow_{\mathcal{R}})$, where $\mathcal{T}$ denotes the set of all terms $\mathcal{T}(\mathcal{F}, \mathcal{V})$. We sometimes say the (innermost) runtime complexity of $\mathcal{R}$ is *linear*, *quadratic*, or *polynomial* if $\mathsf{rc}^{(\mathsf{i})}_{\mathcal{R}}(n)$ is bounded linearly, quadratically, or polynomially in n, respectively. Note that the derivational complexity and the runtime complexity of a TRS $\mathcal{R}$ may be quite different: In general it is not possible to bound $\mathsf{dc}_{\mathcal{R}}$ polynomially in $\mathsf{rc}_{\mathcal{R}}$, as witnessed by Example 1 and the observation that the runtime complexity of $\mathcal{R}$ is linear (see Example 34, below).

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 $t_1 \succ t_2 \succ t_3 \cdots$. A well-founded proper order that is also a rewrite relation is called a *reduction order*. We say a reduction order $\succ$ and a TRS $\mathcal{R}$ are *compatible* if $\mathcal{R} \subseteq \succ$. It is well-known that a TRS is terminating if and only if there exists a compatible reduction order. An $\mathcal{F}$-*algebra* $\mathcal{A}$ consists of a carrier set A and a collection of interpretations $f_{\mathcal{A}}$ 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 partial order on A such that every $f_{\mathcal{A}}$ is monotone in all arguments. An *assignment* $\alpha \colon \mathcal{V} \rightarrow A$ is a function mapping variables to elements in the carrier. 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 $\alpha \colon \mathcal{V} \rightarrow A$.

3 The Dependency Pair Method

The purpose of this section is to take a fresh look at the dependency pair method from the point of complexity analysis. Familiarity with [7,9] will be helpful. 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 complexity analysis we employ a similar observation: From a given term t one can extract a list of subterms whose sum of the derivation lengths is equal to the derivational length of t.

Let X be a set of symbols. We write $C\langle t_1, \ldots, t_n \rangle_X$ to denote $C[t_1, \ldots, t_n]$, whenever $\mathsf{root}(t_i) \in X$ for all $1 \leqslant i \leqslant n$ and C is an n-hole context containing no X-symbols. (Note that the context C may be degenerate and doesn't contain a hole $\Box$ or it may be that C is a hole.) Then, every term t can be uniquely written in the form $C\langle t_1, \ldots, t_n \rangle_X$.

Lemma 2. *Let t be a terminating term, and let σ be a substitution. Then* $\mathsf{dl}(t\sigma, \rightarrow_{\mathcal{R}}) = \sum_{1 \leqslant i \leqslant n} \mathsf{dl}(t_i\sigma, \rightarrow_{\mathcal{R}})$, *whenever* $t = C\langle t_1, \ldots, t_n \rangle_{\mathcal{D} \cup \mathcal{V}}$.

[1] We can replace $\mathcal{T}_{\mathsf{b}}$ by the set of terms $f(t_1, \ldots, t_n)$ with $f \in \mathcal{D}$, whose arguments t_i are in normal form, while keeping all results in this paper.

We define the function COM as a mapping from tuples of terms to terms as follows: $\textsc{com}(t_1, \ldots, t_n)$ is t_1 if $n = 1$, and $c(t_1, \ldots, t_n)$ otherwise. Here c is a fresh n-ary function symbol called *compound symbol*. The above lemma motivates the next definition of *weak dependency pairs*.

Definition 3. *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 $\mathcal{F}^\sharp = \mathcal{F} \cup \{f^\sharp \mid f \in \mathcal{F}\}$. 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 $l^\sharp \to \textsc{com}(u_1^\sharp, \ldots, u_n^\sharp)$ is called a* weak dependency pair *of $\mathcal{R}$. 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 four weak dependency pairs:

$$\begin{array}{llll} 5\colon & x -^\sharp \mathsf{0} \to x & 7\colon & \mathsf{0} \div^\sharp \mathsf{s}(y) \to \mathsf{c}_1 \\ 6\colon & \mathsf{s}(x) -^\sharp \mathsf{s}(y) \to x -^\sharp y & 8\colon & \mathsf{s}(x) \div^\sharp \mathsf{s}(y) \to (x - y) \div^\sharp \mathsf{s}(y) \end{array}$$

Lemma 5. *Let $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ be a terminating term with $\mathsf{root}(t) \in \mathcal{D}$. We have $\mathsf{dl}(t, \to_{\mathcal{R}}) = \mathsf{dl}(t^\sharp, \to_{\mathsf{WDP}(\mathcal{R})\cup\mathcal{R}})$.*

Proof. We show $\mathsf{dl}(t, \to_{\mathcal{R}}) \leqslant \mathsf{dl}(t^\sharp, \to_{\mathsf{WDP}(\mathcal{R})\cup\mathcal{R}})$ by induction on $\ell = \mathsf{dl}(t, \to_{\mathcal{R}})$. If $\ell = 0$, the inequality is trivial. Suppose $\ell > 0$. Then there exists a term u such that $t \to_{\mathcal{R}} u$ and $\mathsf{dl}(u, \to_{\mathcal{R}}) = \ell - 1$. We distinguish two cases depending on the rewrite position p.

– If p is a position below the root, then clearly $\mathsf{root}(u) = \mathsf{root}(t) \in \mathcal{D}$ and $t^\sharp \to_{\mathcal{R}} u^\sharp$. The induction hypothesis yields $\mathsf{dl}(u, \to_{\mathcal{R}}) \leqslant \mathsf{dl}(u^\sharp, \to_{\mathsf{WDP}(\mathcal{R})\cup\mathcal{R}})$, and we obtain $\ell \leqslant \mathsf{dl}(t^\sharp, \to_{\mathsf{WDP}(\mathcal{R})\cup\mathcal{R}})$.
– If p is a root position, then there exist a rewrite rule $l \to r \in \mathcal{R}$ and a substitution σ such that $t = l\sigma$ and $u = r\sigma$. We have $r = C\langle u_1, \ldots, u_n\rangle_{\mathcal{D}\cup\mathcal{V}}$ and thus by definition $l^\sharp \to \textsc{com}(u_1^\sharp, \ldots, u_n^\sharp) \in \mathsf{WDP}(\mathcal{R})$ such that $t^\sharp = l^\sharp\sigma$. Now, either $u_i \in \mathcal{V}$ or $\mathsf{root}(u_i) \in \mathcal{D}$ for every $1 \leqslant i \leqslant n$. Suppose $u_i \in \mathcal{V}$. Then $u_i^\sharp\sigma = u_i\sigma$ and clearly no dependency pair symbol can occur and thus,

$$\mathsf{dl}(u_i\sigma, \to_{\mathcal{R}}) = \mathsf{dl}(u_i^\sharp\sigma, \to_{\mathcal{R}}) = \mathsf{dl}(u_i^\sharp\sigma, \to_{\mathsf{WDP}(\mathcal{R})\cup\mathcal{R}}) .$$

Otherwise, $\mathsf{root}(u_i) \in \mathcal{D}$ and thus $u_i^\sharp\sigma = (u_i\sigma)^\sharp$. We have $\mathsf{dl}(u_i\sigma, \to_{\mathcal{R}}) \leqslant \mathsf{dl}(u, \to_{\mathcal{R}}) < l$, and conclude $\mathsf{dl}(u_i\sigma, \to_{\mathcal{R}}) \leqslant \mathsf{dl}(u_i^\sharp\sigma, \to_{\mathsf{WDP}(\mathcal{R})\cup\mathcal{R}})$ by the induction hypothesis. Therefore,

$$\begin{aligned} \ell = \mathsf{dl}(u, \to_{\mathcal{R}}) + 1 &= \sum_{1\leqslant i\leqslant n} \mathsf{dl}(u_i\sigma, \mathcal{R}) + 1 \leqslant \sum_{1\leqslant i\leqslant n} \mathsf{dl}(u_i^\sharp\sigma, \mathsf{WDP}(\mathcal{R}) \cup \mathcal{R}) + 1 \\ &\leqslant \mathsf{dl}(\textsc{com}(u_1^\sharp, \ldots, u_n^\sharp)\sigma, \to_{\mathsf{WDP}(\mathcal{R})\cup\mathcal{R}}) + 1 = \mathsf{dl}(t^\sharp, \to_{\mathsf{WDP}(\mathcal{R})\cup\mathcal{R}}) . \end{aligned}$$

Here we used Lemma 2 for the second equality.

Note that t is $\mathcal{R}$-reducible if and only if $t^\sharp$ is $\mathsf{WDP}(\mathcal{R}) \cup \mathcal{R}$-reducible. Hence as t is terminating, $t^\sharp$ is terminating on $\to_{\mathsf{WDP}(\mathcal{R})\cup\mathcal{R}}$. Thus, similarly, $\mathsf{dl}(t, \to_\mathcal{R}) \geqslant \mathsf{dl}(t^\sharp, \to_{\mathsf{WDP}(\mathcal{R})\cup\mathcal{R}})$ is shown by induction on $\mathsf{dl}(t^\sharp, \to_{\mathsf{WDP}(\mathcal{R})\cup\mathcal{R}})$. □

Lemma 6. *Let t be a terminating term and σ a substitution such that $x\sigma$ is a normal form of $\mathcal{R}$ for all $x \in \mathcal{V}\mathsf{ar}(t)$. Then $\mathsf{dl}(t\sigma, \to_\mathcal{R}) = \sum_{1 \leqslant i \leqslant n} \mathsf{dl}(t_i\sigma, \to_\mathcal{R})$, whenever $t = C\langle t_1, \dots, t_n\rangle_\mathcal{D}$.*

Definition 7. *Let $\mathcal{R}$ be a TRS. If $l \to r \in \mathcal{R}$ and $r = C\langle u_1, \dots, u_n\rangle_\mathcal{D}$ then the rewrite rule $l^\sharp \to \textsc{com}(u_1^\sharp, \dots, u_n^\sharp)$ is called a* weak innermost dependency pair *of $\mathcal{R}$. The set of all weak innermost dependency pairs is denoted by* $\mathsf{WIDP}(\mathcal{R})$*.*

Example 8 (continued from Example 1). The set $\mathsf{WIDP}(\mathcal{R})$ consists of the next four weak dependency pairs (with respect to $\xrightarrow{\mathrm{i}}$):

$$x -^\sharp \mathsf{0} \to \mathsf{c}_1 \qquad \mathsf{0} \div^\sharp y \to \mathsf{c}_2$$
$$\mathsf{s}(x) -^\sharp \mathsf{s}(y) \to x -^\sharp y \qquad \mathsf{s}(x) \div^\sharp \mathsf{s}(y) \to (x - y) \div^\sharp \mathsf{s}(y)$$

The next lemma adapts Lemma 5 to innermost rewriting.

Lemma 9. *Let t be an innermost terminating term in $\mathcal{T}(\mathcal{F}, \mathcal{V})$ with $\mathsf{root}(t) \in \mathcal{D}$. We have $\mathsf{dl}(t, \xrightarrow{i}_\mathcal{R}) = \mathsf{dl}(t^\sharp, \xrightarrow{i}_{\mathsf{WIDP}(\mathcal{R})\cup\mathcal{R}})$.*

We conclude this section by discussing the applicability of standard dependency pairs ([7]) in complexity analysis. For that we recall the standard definition of dependency pairs.

Definition 10 ([7]). *The set $\mathsf{DP}(\mathcal{R})$ of (standard)* dependency pairs *of a TRS $\mathcal{R}$ is defined as $\{l^\sharp \to u^\sharp \mid l \to r \in \mathcal{R}, u \trianglelefteq r, \mathsf{root}(u) \in \mathcal{D}\}$.*

The next example shows that Lemma 5 (Lemma 9) does not hold if we replace weak (innermost) dependency pairs with standard dependency pairs.

Example 11. Consider the one-rule TRS $\mathcal{R}$: $\mathsf{f}(\mathsf{s}(x)) \to \mathsf{g}(\mathsf{f}(x), \mathsf{f}(x))$. $\mathsf{DP}(\mathcal{R})$ is the singleton of $\mathsf{f}^\sharp(\mathsf{s}(x)) \to \mathsf{f}^\sharp(x)$. Let $t_n = \mathsf{f}(\mathsf{s}^n(x))$ for each $n \geqslant 0$. Since $t_{n+1} \to_\mathcal{R} \mathsf{g}(t_n, t_n)$ holds for all $n \geqslant 0$, it is easy to see $\mathsf{dl}(t_{n+1}, \to_\mathcal{R}) \geqslant 2^n$, while $\mathsf{dl}(t_{n+1}^\sharp, \to_{\mathsf{DP}(\mathcal{R})\cup\mathcal{R}}) = n$.

Hence, in general we cannot replace weak dependency pairs with (standard) dependency pairs. However, if we restrict our attention to innermost rewriting, we can employ dependency pairs in complexity analysis without introducing fallacies, when specific conditions are met.

Lemma 12. *Let t be an innermost terminating term with $\mathsf{root}(t) \in \mathcal{D}$. If all compound symbols in $\mathsf{WIDP}(\mathcal{R})$ are nullary, $\mathsf{dl}(t, \xrightarrow{i}_\mathcal{R}) \leqslant \mathsf{dl}(t^\sharp, \xrightarrow{i}_{\mathsf{DP}(\mathcal{R})\cup\mathcal{R}}) + 1$ holds.*

Example 13 (continued from Example 8). The occurring compound symbols are nullary. $\mathsf{DP}(\mathcal{R})$ consists of the next three dependency pairs:

$$\mathsf{s}(x) -^\sharp \mathsf{s}(y) \to x -^\sharp y \qquad \mathsf{s}(x) \div^\sharp \mathsf{s}(y) \to x -^\sharp y$$
$$\mathsf{s}(x) \div^\sharp \mathsf{s}(y) \to (x - y) \div^\sharp \mathsf{s}(y)$$

4 Usable Rules

In the previous section, we studied the dependency pair method in the light of complexity analysis. Let $\mathcal{R}$ be a TRS and $\mathcal{P}$ a set of weak dependency pairs, weak innermost dependency pairs, or standard dependency pairs of $\mathcal{R}$. Lemmata 5, 9, and 12 describe a strong connection between the length of derivations in the original TRSs $\mathcal{R}$ and the transformed (and extended) system $\mathcal{P} \cup \mathcal{R}$. In this section we show how we can simplify the new TRS $\mathcal{P} \cup \mathcal{R}$ by employing *usable rules*.

Definition 14. *We write $f \rhd_{\mathrm{d}} g$ if there exists a rewrite rule $l \to r \in \mathcal{R}$ such that $f = \mathsf{root}(l)$ and g is a defined function symbol in $\mathcal{F}\mathsf{un}(r)$. For a set $\mathcal{G}$ of defined function symbols we denote by $\mathcal{R}{\restriction}\mathcal{G}$ the set of rewrite rules $l \to r \in \mathcal{R}$ with $\mathsf{root}(l) \in \mathcal{G}$. The set $\mathcal{U}(t)$ of usable rules of a term t is defined as $\mathcal{R}{\restriction}\{g \mid f \rhd_{\mathrm{d}}^* g$ for some $f \in \mathcal{F}\mathsf{un}(t)\}$. Finally, if $\mathcal{P}$ is a set of (weak) dependency pairs then $\mathcal{U}(\mathcal{P}) = \bigcup_{l \to r \in \mathcal{P}} \mathcal{U}(r)$.*

Example 15 (continued from Examples 4 and 8). The sets of usable rules are equal for the weak dependency pairs and for the weak innermost dependency pairs, i.e., we have $\mathcal{U}(\mathsf{WDP}(\mathcal{R})) = \mathcal{U}(\mathsf{WIDP}(\mathcal{R})) = \{1, 2\}$.

The usable rule criterion in termination analysis (cf. [12,9]) asserts that a non-terminating rewrite sequence of $\mathcal{R} \cup \mathsf{DP}(\mathcal{R})$ can be transformed into a non-terminating rewrite sequence of $\mathcal{U}(\mathsf{DP}(\mathcal{R})) \cup \mathsf{DP}(\mathcal{R}) \cup \{\mathsf{g}(x, y) \to x, \mathsf{g}(x, y) \to y\}$, where g is a fresh function symbol. Because $\mathcal{U}(\mathsf{DP}(\mathcal{R}))$ is a (small) subset of $\mathcal{R}$ and most termination methods can handle $\mathsf{g}(x, y) \to x$ and $\mathsf{g}(x, y) \to y$ easily, the termination analysis often becomes easy by switching the target of analysis from the former TRS to the latter TRS. Unfortunately the transformation used in [12,9] increases the size of starting terms, therefore we cannot adopt this transformation approach. Note, however that the usable rule criteria for innermost termination [8] can be directly applied in the context of complexity analysis. Nevertheless, one may show a new type of usable rule criterion by exploiting the basic property of a starting term. Recall that $\mathcal{T}_{\mathsf{b}}$ denotes the set of basic terms; we set $\mathcal{T}_{\mathsf{b}}^{\sharp} = \{t^{\sharp} \mid t \in \mathcal{T}_{\mathsf{b}}\}$.

Lemma 16. *Let $\mathcal{P}$ be a set of (weak) dependency pairs and let $(t_i)_{i=0,1,\ldots}$ be a (finite or infinite) derivation of $\mathcal{R} \cup \mathcal{P}$. If $t_0 \in \mathcal{T}_{\mathsf{b}}^{\sharp}$ then $(t_i)_{i=0,1,\ldots}$ is a derivation of $\mathcal{U}(\mathcal{P}) \cup \mathcal{P}$.*

Proof. Let $\mathcal{G}$ be the set of all non-usable symbols with respect to $\mathcal{P}$. We write $P(t)$ if $t|_q \in \mathcal{NF}(\mathcal{R})$ for all $q \in \mathcal{P}\mathsf{os}_{\mathcal{G}}(t)$. Since $t_i \to_{\mathcal{U}(\mathcal{P}) \cup \mathcal{P}} t_{i+1}$ holds whenever $P(t_i)$ and $t_i \to_{\mathcal{R} \cup \mathcal{P}} t_{i+1}$, it is sufficient to show $P(t_i)$ for all i. We perform induction on i.

1. Assume $i = 0$. Since $t_0 \in \mathcal{T}_{\mathsf{b}}^{\sharp}$, we have $t_0 \in \mathcal{NF}(\mathcal{R})$ and thus $t|_p \in \mathcal{NF}(\mathcal{R})$ for all positions p. The assertion P follows trivially.
2. Suppose $i > 0$. By induction hypothesis, there exist $l \to r \in \mathcal{U}(\mathcal{P}) \cup \mathcal{P}$, $p \in \mathcal{P}\mathsf{os}(t_{i-1})$, and a substitution σ such that $t_{i-1}|_p = l\sigma$ and $t_i|_p = r\sigma$. In order to show property P for t_i, we fix a position $q \in \mathcal{G}$. We have to show $t_i|_q \in \mathcal{NF}(\mathcal{R})$. We distinguish three cases:

– Suppose that q is above p. Then $t_{i-1}|_q$ is reducible, but this contradicts the induction hypothesis $P(t_{i-1})$.
– Suppose p and q are parallel but distinct. Since $t_{i-1}|_q = t_i|_q \in \mathcal{NF}(\mathcal{R})$ holds, we obtain $P(t_i)$.
– Otherwise, q is below p. Then, $t_i|_q$ is a subterm of $r\sigma$. Because r contains no $\mathcal{G}$-symbols by the definition of usable symbols, $t_i|_q$ is a subterm of $x\sigma$ for some $x \in \mathcal{V}\mathsf{ar}(r) \subseteq \mathcal{V}\mathsf{ar}(l)$. Therefore, $t_i|_q$ is also a subterm of t_{i-1}, from which $t_i|_q \in \mathcal{NF}(\mathcal{R})$ follows. We obtain $P(t_i)$. □

The following theorem follows from Lemmata 5, 9, and 12 in conjunction with the above Lemma 16. It adapts the usable rule criteria to complexity analysis.[2]

Theorem 17. *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 $\mathcal{P} = \mathsf{WIDP}(\mathcal{R})$. Moreover, suppose all compound symbols in $\mathsf{WIDP}(\mathcal{R})$ are nullary then $\mathsf{dl}(t, \xrightarrow{i}_\mathcal{R}) \leqslant \mathsf{dl}(t^\sharp, \to_{\mathcal{U}(\mathsf{DP}(\mathcal{R})) \cup \mathsf{DP}(\mathcal{R})}) + 1$.*

It is worth stressing that it is (often) easier to analyse the complexity of $\mathcal{U}(\mathcal{P}) \cup \mathcal{P}$ than the complexity of $\mathcal{R}$. To clarify the applicability of the theorem in complexity analysis, we consider two restrictive classes of polynomial interpretations, whose definitions are motivated by [13].

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 a linear polynomial. 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. 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 also [13]).

Corollary 18. *Let $\mathcal{R}$ be a TRS and let $\mathcal{P} = \mathsf{WDP}(\mathcal{R})$ or $\mathcal{P} = \mathsf{WIDP}(\mathcal{R})$. If $\mathcal{U}(\mathcal{P}) \cup \mathcal{P}$ is compatible with a linear or quadratic restricted interpretation, the (innermost) runtime complexity function $\mathsf{rc}^{(\mathsf{i})}_\mathcal{R}$ with respect to $\mathcal{R}$ is linear or quadratic, respectively. Moreover, suppose all compound symbols in $\mathsf{WIDP}(\mathcal{R})$ are nullary and $\mathcal{U}(\mathsf{DP}(\mathcal{R})) \cup \mathsf{DP}(\mathcal{R})$ is compatible with a linear (quadratic) restricted interpretation, then $\mathcal{R}$ admits at most linear (quadratic) innermost runtime complexity.*

Proof. Let $\mathcal{R}$ be a TRS. For simplicity we suppose $\mathcal{P} = \mathsf{WDP}(\mathcal{R})$ and assume the existence of a linear restricted interpretation $\mathcal{A}$, compatible with $\mathcal{U}(\mathcal{P}) \cup \mathcal{P}$. Clearly this implies the well-foundedness of the relation $\to_{\mathcal{U}(\mathcal{P}) \cup \mathcal{P}}$, which in turn implies the well-foundedness of $\to_\mathcal{R}$, cf. Lemma 16. Hence Theorem 17 is applicable and we conclude $\mathsf{dl}(t, \to_\mathcal{R}) \leqslant \mathsf{dl}(t^\sharp, \to_{\mathsf{WDP}(\mathcal{R}) \cup \mathcal{R}})$. On the other hand,

[2] Note that Theorem 17 only holds for *basic* terms $t \in \mathcal{T}_\mathsf{b}^\sharp$. In order to show this, we need some additional technical lemmas, which are the subject of the next section.

compatibility with $\mathcal{A}$ implies that $\mathsf{dl}(t^\sharp, \to_{\mathsf{WDP}(\mathcal{R}) \cup \mathcal{R}}) = \mathcal{O}(|t^\sharp|)$. As $|t^\sharp| = |t|$, we can combine these equalities to conclude linear runtime complexity of $\mathcal{R}$. □

5 The Weight Gap Principle

We recall the notion of *relative rewriting* ([14,11]).

Definition 19. *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 $\mathcal{S}$.*[3]

Since $\mathsf{dl}(t, \to_{\mathcal{R}/\mathcal{S}})$ corresponds to the number of $\to_{\mathcal{R}}$-steps in a maximal derivation of $\to_{\mathcal{R}\cup\mathcal{S}}$ from t, we easily see the bound $\mathsf{dl}(t, \to_{\mathcal{R}/\mathcal{S}}) \leqslant \mathsf{dl}(t, \to_{\mathcal{R}\cup\mathcal{S}})$. In this section we study this opposite, i.e., we figure out a way to give an upper-bound of $\mathsf{dl}(t, \to_{\mathcal{R}\cup\mathcal{S}})$ by a function of $\mathsf{dl}(t, \to_{\mathcal{R}/\mathcal{S}})$.

First we introduce the key ingredient, *strongly linear interpretations*, a very restrictive form of polynomial interpretations. Let $\mathcal{F}$ denote a signature. A *strongly linear interpretation* (*SLI* for short) is a WMA $(\mathcal{A}, \succ)$ that satisfies the following properties: (i) the carrier of $\mathcal{A}$ is the set of natural numbers $\mathbb{N}$, (ii) all interpretation functions $f_{\mathcal{A}}$ are strongly linear, (iii) the proper order $\succ$ is the standard order $>$ on $\mathbb{N}$. Note that an SLI $\mathcal{A}$ is conceivable as a weight function. We define the maximum weight $\mathsf{M}_{\mathcal{A}}$ of $\mathcal{A}$ as $\max\{f_{\mathcal{A}}(0, \ldots, 0) \mid f \in \mathcal{F}\}$. Let $\mathcal{A}$ denote an SLI, let α_0 denote the assignment mapping any variable to 0, i.e., $\alpha_0(x) = 0$ for all $x \in \mathcal{V}$, and let t be a term. We write $[t]$ as an abbreviation for $[\alpha_0]_{\mathcal{A}}(t)$.

Lemma 20. *Let $\mathcal{A}$ be an SLI and let t be a term. Then $[t] \leqslant \mathsf{M}_{\mathcal{A}} \cdot |t|$ holds.*

Proof. By induction on t. If $t \in \mathcal{V}$ then $[t] = 0 \leqslant \mathsf{M}_{\mathcal{A}} \cdot |t|$. Otherwise, suppose $t = f(t_1, \ldots, t_n)$, where $f_{\mathcal{A}}(x_1, \ldots, x_n) = x_1 + \ldots + x_n + c$. By the induction hypothesis and $c \leqslant \mathsf{M}_{\mathcal{A}}$ we obtain the following inequalities:

$$\begin{aligned} [t] &= f_{\mathcal{A}}([t_1], \ldots, [t_n]) \leqslant [t_1] + \cdots + [t_n] + c \\ &\leqslant \mathsf{M}_{\mathcal{A}} \cdot |t_1| + \cdots + \mathsf{M}_{\mathcal{A}} \cdot |t_n| + \mathsf{M}_{\mathcal{A}} = \mathsf{M}_{\mathcal{A}} \cdot |t| \ . \end{aligned}$$ □

The conception of strongly linear interpretations as weight functions allows us to study (possible) weight increase throughout a rewrite derivation. This observation is reflected in the next definition.

Definition 21. *Let $\mathcal{A}$ be an algebra and let $\mathcal{R}$ be a TRS. The* weight gap $\Delta(\mathcal{A}, \mathcal{R})$ of $\mathcal{A}$ with respect to $\mathcal{R}$ *is defined on* $\mathbb{N}$ *as follows:* $\Delta(\mathcal{A}, \mathcal{R}) = \max\{[r] \dot{-} [l] \mid l \to r \in \mathcal{R}\}$*, where $\dot{-}$ is defined as usual:* $m \dot{-} n := \max\{m - n, 0\}$

The following *weight gap principle* is a direct consequence of the definitions.

Lemma 22. *Let $\mathcal{R}$ be a TRS and $\mathcal{A}$ an SLI. If $s \to_{\mathcal{R}} t$ then $[s] + \Delta(\mathcal{A}, \mathcal{R}) \geqslant [t]$.*

We stress that the lemma does not require any condition. Indeed, the implication in the lemma holds even if TRS $\mathcal{R}$ is *not* compatible with a strongly linear interpretation. This principle brings us to the next theorem.

[3] Note that $\to_{\mathcal{R}/\mathcal{S}} = \to_{\mathcal{R}}$, if $\mathcal{S} = \varnothing$.

Theorem 23. *Let $\mathcal{R}$ and $\mathcal{S}$ be TRSs, and $\mathcal{A}$ an SLI compatible with $\mathcal{S}$. Then we have $\mathsf{dl}(t, \to_{\mathcal{R} \cup \mathcal{S}}) \leqslant (1 + \Delta(\mathcal{A}, \mathcal{R})) \cdot \mathsf{dl}(t, \to_{\mathcal{R}/\mathcal{S}}) + \mathsf{M}_{\mathcal{A}} \cdot |t|$, whenever t is terminating on $\mathcal{R} \cup \mathcal{S}$.*

Proof. Let $m = \mathsf{dl}(t, \to_{\mathcal{R}/\mathcal{S}})$, let $n = |t|$, and set $\Delta = \Delta(\mathcal{A}, \mathcal{R})$. Any derivation of $\to_{\mathcal{R} \cup \mathcal{S}}$ is representable as follows

$$s_0 \to_{\mathcal{S}}^{k_0} t_0 \to_{\mathcal{R}} s_1 \to_{\mathcal{S}}^{k_1} t_1 \to_{\mathcal{R}} \cdots \to_{\mathcal{S}}^{k_m} t_m \ ,$$

and without loss of generality we may assume that the derivation is maximal. We observe the next two facts.

(a) $k_i \leqslant [s_i] - [t_i]$ holds for all $0 \leqslant i \leqslant m$. This is because $[s] \geqslant [t] + 1$ whenever $s \to_{\mathcal{S}} t$ by the assumption $\mathcal{S} \subseteq >_{\mathcal{A}}$, and we have $s_i \to_{\mathcal{S}}{}^{k_i} t_i$.
(b) $[s_{i+1}] - [t_i] \leqslant \Delta$ holds for all $0 \leqslant i < m$ as due to Lemma 22 we have $[t_i] + \Delta \geqslant [s_{i+1}]$.

We obtain the following inequalities:

$$\begin{aligned}
\mathsf{dl}(s_0, \to_{\mathcal{R} \cup \mathcal{S}}) &= m + k_0 + \cdots + k_m \\
&\leqslant m + ([s_0] - [t_0]) + \cdots + ([s_m] - [t_m]) \\
&= m + [s_0] + ([s_1] - [t_0]) + \cdots + ([s_m] - [t_{m-1}]) - [t_m] \\
&\leqslant m + [s_0] + m\Delta - [t_m] \\
&\leqslant m + [s_0] + m\Delta \\
&\leqslant m + \mathsf{M}_{\mathcal{A}} \cdot n + m\Delta = (1 + \Delta)m + \mathsf{M}_{\mathcal{A}} \cdot n \ .
\end{aligned}$$

Here we used (a) m-times in the second line, (b) $m - 1$-times in the fourth line, and Lemma 20 in the last line. □

The next example clarifies that the conditions expressed in Theorem 23 are optimal: We cannot replace the assumption that the algebra $\mathcal{A}$ is *strongly* linear with a weaker assumption: Already if $\mathcal{A}$ is a linear polynomial interpretation, the derivation height of $\mathcal{R} \cup \mathcal{S}$ cannot be bounded *polynomially* in $\mathsf{dl}(t, \to_{\mathcal{R}/\mathcal{S}})$ and $|t|$ alone.

Example 24. Consider the TRSs $\mathcal{R}$

$$\begin{aligned}
\mathsf{exp}(\mathsf{0}) &\to \mathsf{s}(\mathsf{0}) & \mathsf{d}(\mathsf{0}) &\to \mathsf{0} \\
\mathsf{exp}(\mathsf{r}(x)) &\to \mathsf{d}(\mathsf{exp}(x)) & \mathsf{d}(\mathsf{s}(x)) &\to \mathsf{s}(\mathsf{s}(\mathsf{d}(x)))
\end{aligned}$$

This TRS formalises the exponentiation function. Setting $t_n = \mathsf{exp}(\mathsf{r}^n(\mathsf{0}))$ we obtain $\mathsf{dl}(t_n, \to_{\mathcal{R}}) \geqslant 2^n$ for each $n \geqslant 0$. Thus the runtime complexity of $\mathcal{R}$ is (at least) exponential. In order to show the claim, we split $\mathcal{R}$ into two TRSs $\mathcal{R}_1 = \{\mathsf{exp}(\mathsf{0}) \to \mathsf{s}(\mathsf{0}), \mathsf{exp}(\mathsf{r}(x)) \to \mathsf{d}(\mathsf{exp}(x))\}$ and $\mathcal{R}_2 = \{\mathsf{d}(\mathsf{0}) \to \mathsf{0}, \mathsf{d}(\mathsf{s}(x)) \to \mathsf{s}(\mathsf{s}(\mathsf{d}(x)))\}$. Then it is easy to verify that the next linear polynomial interpretation $\mathcal{A}$ is compatible with $\mathcal{R}_2$: $\mathsf{0}_{\mathcal{A}} = 0$, $\mathsf{d}_{\mathcal{A}}(x) = 3x$, and $\mathsf{s}_{\mathcal{A}}(x) = x + 1$. Moreover an upper-bound of $\mathsf{dl}(t_n, \to_{\mathcal{R}_1/\mathcal{R}_2})$ can be estimated by using the following polynomial interpretation $\mathcal{B}$: $0_{\mathcal{B}} = 0$, $\mathsf{d}_{\mathcal{B}}(x) = \mathsf{s}_{\mathcal{B}}(x) = x$, and $\mathsf{exp}_{\mathcal{B}}(x) = \mathsf{r}_{\mathcal{B}}(x) = x + 1$. Since $\rightarrow_{\mathcal{R}_1} \subseteq >_{\mathcal{B}}$ and $\rightarrow^*_{\mathcal{R}_2} \subseteq \geqslant_{\mathcal{B}}$ hold, we have $\rightarrow_{\mathcal{R}_1/\mathcal{R}_2} \subseteq >_{\mathcal{B}}$. Hence $\mathsf{dl}(t_n, \rightarrow_{\mathcal{R}_1/\mathcal{R}_2}) \leqslant [\alpha_0]_{\mathcal{B}}(t_n) = n + 2$. But clearly from this we cannot conclude a polynomial bound on the derivation length of $\mathcal{R}_1 \cup \mathcal{R}_2 = \mathcal{R}$, as the runtime complexity of $\mathcal{R}$ is exponential, at least.

To conclude this section, we show that Theorem 17 can only hold for *basic* terms $t \in \mathcal{T}_{\mathsf{b}}^{\sharp}$.

Example 25. Consider the one-rule TRS $\mathcal{R} = \{\mathsf{a}(\mathsf{b}(x)) \rightarrow \mathsf{b}(\mathsf{b}(\mathsf{a}(x)))\}$ from [15, Example 2.50]. It is not difficult to see that $\mathsf{dl}(\mathsf{a}^n(\mathsf{b}(x)), \rightarrow_{\mathcal{R}}) = 2^n - 1$, see [4]. The set $\mathsf{WDP}(\mathcal{R})$ consists of just one dependency pair $\mathsf{a}^{\sharp}(\mathsf{b}(x)) \rightarrow \mathsf{a}^{\sharp}(x))$. In particular the set of usable rules is empty. The following SLI $\mathcal{A}$ is compatible with $\mathsf{WDP}(\mathcal{R})$: $\mathsf{a}^{\sharp}_{\mathcal{A}}(x) = \mathsf{a}_{\mathcal{A}}(x) = x$ and $\mathsf{b}_{\mathcal{A}}(x) = 1$. Hence, due to Lemma 20 we can conclude the existence of a constant K such that $\mathsf{dl}(t^{\sharp}, \rightarrow_{\mathsf{WDP}(\mathcal{R})}) \leqslant K \cdot |t|$. Due to Theorem 17 we conclude linear runtime complexity of $\mathcal{R}$.

6 Reduction Pairs and Argument Filterings

In this section we study the consequences of combining Theorem 17 and Theorem 23. In doing so, we adapt reduction pairs and argument filterings ([7]) to runtime complexity analysis. Let $\mathcal{R}$ be a TRS, and let $\mathcal{A}$ be a strongly linear interpretation and suppose we consider weak, weak innermost, or (standard) dependency pairs $\mathcal{P}$. If $\mathcal{U}(\mathcal{P}) \subseteq >_{\mathcal{A}}$ then there exist constants $K, L \geqslant 0$ (depending on $\mathcal{P}$ and $\mathcal{A}$ only) such that

$$\mathsf{dl}(t, \rightarrow_{\mathcal{R}}) \leqslant K \cdot \mathsf{dl}(t^{\sharp}, \rightarrow_{\mathcal{P}/\mathcal{U}(\mathcal{P})}) + L \cdot |t^{\sharp}| \ ,$$

for all terminating basic terms $t \in \mathcal{T}_{\mathsf{b}}$. This follows from the combination of Theorems 17 and 23. Thus, in order to estimate the derivation length of t with respect to $\mathcal{R}$ it suffices to estimate the maximal $\mathcal{P}$ steps, i.e., we have to estimate $\mathsf{dl}(t^{\sharp}, \rightarrow_{\mathcal{P}/\mathcal{U}(\mathcal{P})})$ suitably. Consider a maximal derivation $(t_i)_{i=0,\ldots,n}$ of $\rightarrow_{\mathcal{P}/\mathcal{U}(\mathcal{P})}$ with $t_0 = t^{\sharp}$. For every $0 \leqslant i < n$ there exist terms u_i and v_i such that

$$t_i \rightarrow^*_{\mathcal{U}(\mathcal{P})} u_i \rightarrow_{\mathcal{P}} v_i \rightarrow^*_{\mathcal{U}(\mathcal{P})} t_{i+1} \ . \tag{1}$$

Let $\gtrsim$ and $\succ$ be a pair of orders with $\gtrsim \cdot \succ \cdot \gtrsim \subseteq \succ$. If $t_i \gtrsim u_i \succ v_i \gtrsim t_{i+1}$ holds for all $0 \leqslant i < n$, we obtain $t^{\sharp} = t_0 \succ t_1 \succ \cdots \succ t_n$. Therefore, $\mathsf{dl}(t^{\sharp}, \rightarrow_{\mathcal{P}/\mathcal{U}(\mathcal{P})})$ can be bounded in the maximal length of $\succ$-descending steps. We formalise these observations through the use of *reduction pairs* and *collapsible orders.*

Definition 26. *Let $\mathcal{R}$ be a TRS, let $\mathcal{P}$ a set of weak 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 $s \rightarrow_{\mathcal{P} \cup \mathcal{U}(\mathcal{P})} t$ 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 $\mathcal{R}$.*

Note that most reduction orders are collapsible. For instance, if $\mathcal{A}$ is a polynomial interpretation then $>_{\mathcal{A}}$ is collapsible, as witnessed by the evaluation function $[\alpha_0]_{\mathcal{A}}$. Furthermore, simplification orders like MPO, LPO and KBO are collapsible (cf. [2,3,5]).[4]

Definition 27. *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* $\gtrsim \cdot \succ \cdot \gtrsim \subseteq \succ$. *A reduction pair* $(\gtrsim, \succ)$ *is called* collapsible *for a TRS* $\mathcal{R}$ *if* $\succ$ *is collapsible for* $\mathcal{R}$.

Recall the derivation in (1): Due to compound symbols the rewrite step $u_i \to_{\mathcal{P}} v_i$ may take place below the root. Hence $\mathcal{P} \subseteq \succ$ does not ensure $u_i \succ v_i$. To address this problem we introduce a notion of *safety* that is based on the next definitions.

Definition 28. *The set* $\mathcal{T}_{\mathsf{c}}^{\sharp}$ *is inductively defined as follows (i)* $\mathcal{T}_{\mathsf{b}}^{\sharp} \subseteq \mathcal{T}_{\mathsf{c}}^{\sharp}$ *and (ii)* $c(t_1, \ldots, t_n) \in \mathcal{T}_{\mathsf{c}}^{\sharp}$, *whenever* $t_1, \ldots, t_n \in \mathcal{T}_{\mathsf{c}}^{\sharp}$ *and* c *a compound symbol.*

Definition 29. *A proper order* $\succ$ *on* $\mathcal{T}_{\mathsf{c}}^{\sharp}$ *is called* safe *if* $c(s_1, \ldots, s_i, \ldots, s_n) \succ c(s_1, \ldots, t, \ldots, s_n)$ *for all* n*-ary compound symbols* c *and all terms* $s_1, \ldots, s_n, t$ *with* $s_i \succ t$. *A reduction pair* $(\gtrsim, \succ)$ *is called* safe *if* $\succ$ *is safe.*

Lemma 30. *Let* $\mathcal{P}$ *be a set of weak, weak innermost, or standard dependency pairs, and* $(\gtrsim, \succ)$ *be a safe reduction pair such that* $\mathcal{U}(\mathcal{P}) \subseteq \gtrsim$ *and* $\mathcal{P} \subseteq \succ$. *If* $s \in \mathcal{T}_{\mathsf{c}}^{\sharp}$ *and* $s \to_{\mathcal{P}/\mathcal{U}(\mathcal{P})} t$ *then* $s \succ t$ *and* $t \in \mathcal{T}_{\mathsf{c}}^{\sharp}$.

Employing Theorem 17, Theorem 23, and Lemma 30 we arrive at our Main Theorem.

Theorem 31. *Let* $\mathcal{R}$ *be a TRS, let* $\mathcal{A}$ *an SLI, let* $\mathcal{P}$ *be the set of weak, weak innermost, or (standard) 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* $t \in \mathcal{T}_{\mathsf{b}}$, *we have* $\mathsf{dl}(t, \to) \leqslant p(\mathsf{G}(t^{\sharp}, >_{\mathcal{A}}), |t|)$, *where* $p(m, n) := (1 + \Delta(\mathcal{A}, \mathcal{P})) \cdot m + \mathsf{M}_{\mathcal{A}} \cdot n$ *and* $\to$ *denotes* $\to_{\mathcal{R}}$ *or* $\xrightarrow{i}_{\mathcal{R}}$ *depending on whether* $\mathcal{P} = \mathsf{WDP}(\mathcal{R})$ *or* $\mathcal{P} = \mathsf{WIDP}(\mathcal{R})$. *Moreover if all compound symbols in* $\mathsf{WIDP}(\mathcal{R})$ *are nullary we have* $\mathsf{dl}(t, \xrightarrow{i}_{\mathcal{R}}) \leqslant p(\mathsf{G}(t^{\sharp}, >_{\mathcal{A}}), |t|) + 1$.

Proof. First, observe that the assumptions imply that any basic term $t \in \mathcal{T}_{\mathsf{b}}$ is terminating with respect to $\mathcal{R}$. This is a direct consequence of Lemma 16 and Lemma 30 in conjunction with the assumptions of the theorem. Without loss of generality, we assume $\mathcal{P} = \mathsf{WDP}(\mathcal{P})$. By Theorem 17 and 23 we obtain:

$$\begin{aligned}\mathsf{dl}(t, \to) &\leqslant \mathsf{dl}(t^{\sharp}, \to_{\mathcal{U}(\mathcal{P}) \cup \mathcal{P}}) \leqslant p(\mathsf{dl}(t^{\sharp}, \to_{\mathcal{P}/\mathcal{U}(\mathcal{P})}), |t^{\sharp}|) \\ &\leqslant p(\mathsf{G}(t^{\sharp}, >_{\mathcal{A}}), |t^{\sharp}|) = p(\mathsf{G}(t^{\sharp}, >_{\mathcal{A}}), |t|) \ .\end{aligned}$$

In the last line we exploit that $|t^{\sharp}| = |t|$. □

[4] On the other hand it is easy to construct non-collapsible orders: Suppose we extend the natural numbers $\mathbb{N}$ by a non-standard element ∞ such that for any $n \in \mathbb{N}$ we set $\infty > n$. Clearly we cannot collapse ∞ to a natural number.

Note that there exists a subtle disadvantage of Theorem 31 in comparison to Theorem 17. Since the Main Theorem requires compatibility of usable rules with some strongly linear interpretation, all usable rules must be non-duplicating. This is not necessary to meet the requirements of Theorem 17.

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 $\succ$. Another way is to apply an argument filtering to a reduction pair.

Definition 32. *An* argument filtering *for a signature $\mathcal{F}$ is a mapping π that assigns to every n-ary function symbol $f \in \mathcal{F}$ an argument position $i \in \{1, \ldots, n\}$ or a (possibly empty) list $[i_1, \ldots, i_m]$ of argument positions with $1 \leqslant i_1 < \cdots < i_m \leqslant n$. The signature $\mathcal{F}_\pi$ consists of all function symbols f such that $\pi(f)$ is some list $[i_1, \ldots, i_m]$, where in $\mathcal{F}_\pi$ the arity of f is m. Every argument filtering π induces a mapping from $\mathcal{T}(\mathcal{F}, \mathcal{V})$ to $\mathcal{T}(\mathcal{F}_\pi, \mathcal{V})$, also denoted by π:*

$$\pi(t) = \begin{cases} t & \text{if } t \text{ is a variable} \\ \pi(t_i) & \text{if } t = f(t_1, \ldots, t_n) \text{ and } \pi(f) = i \\ f(\pi(t_{i_1}), \ldots, \pi(t_{i_m})) & \text{if } t = f(t_1, \ldots, t_n) \text{ and } \pi(f) = [i_1, \ldots, i_m] \end{cases}$$

An argument filtering π is called safe *if $\pi(c) = [1, \ldots, n]$ for all n-ary compound symbol. For a relation R on $\mathcal{T}(\mathcal{F}, \mathcal{V})$ we define R^π on $\mathcal{T}(\mathcal{F}_\pi, \mathcal{V})$ as follows: $s \mathrel{R^\pi} t$ if and only if $\pi(s) \mathrel{R} \pi(t)$.*

Lemma 33. *If $(\mathcal{A}, \succ)$ is a safe algebra then $(\geqslant_\mathcal{A}, >_\mathcal{A})$ is a safe reduction pair. Furthermore, $(\gtrsim^\pi, \succ^\pi)$ is a safe reduction pair if $(\gtrsim, \succ)$ is a reduction pair and π is a safe argument filtering.*

The below given example applies the Main Theorem to the motivating Example 1 introduced in Section 1.

Example 34 (continued from Example 4). By taking the SLI $\mathcal{A}$ with interpretation $\mathsf{0}_\mathcal{A} = 0$, $\mathsf{s}_\mathcal{A}(x) = x+1$, and $x -_\mathcal{A} y = x+y+1$, we obtain $\mathcal{U}(\mathsf{WDP}(\mathcal{R})) \subseteq >_\mathcal{A}$. Moreover, we take the safe algebra $\mathcal{B}$ with $\mathsf{0}_\mathcal{B} = 0$, $\mathsf{s}_\mathcal{B}(x) = x + 2$, $x -_\mathcal{B} y = x -^\sharp_\mathcal{B} y = x \div^\sharp_\mathcal{B} y = x + 1$, and $(\mathsf{c}_1)_\mathcal{B} = 0$. $\mathcal{B}$ interprets $\mathsf{WDP}(\mathcal{R})$ and $\mathcal{U}(\mathsf{WDP}(\mathcal{R}))$ as follows:

$$\begin{array}{lll} 1\colon\ x+1 \geqslant x & 5\colon\ x+1 > x & 7\colon\quad 1 > 0 \\ 2\colon\ x+3 \geqslant x+1 & 6\colon\ x+3 > x+1 & 8\colon\ x+3 > x+2\,. \end{array}$$

Therefore, $\mathsf{WDP}(\mathcal{R}) \subseteq >_\mathcal{B}$ and $\mathcal{U}(\mathsf{WDP}(\mathcal{R})) \subseteq \geqslant_\mathcal{B}$ hold. Hence, the runtime complexity of $\mathcal{R}$ for full rewriting is linear.

Following the pattern of the proof of Corollary 18 it is an easy exercise to extend Theorem 31 to a method for complexity analysis. In the above example, we have already used the easy fact that (obvious extensions of) linear restricted interpretation may be used as safe reduction pairs.

Corollary 35. *Let $\mathcal{R}$ be a TRS, let $\mathcal{A}$ be an SLI, let $\mathcal{P}$ be the set of weak, weak innermost, or standard dependency pairs, where the compound symbols in* $\mathsf{WIDP}(\mathcal{R})$ *are nullary, if $\mathcal{P} = \mathsf{DP}(\mathcal{R})$. Moreover let $\mathcal{B}$ be a linear or quadratic restricted interpretation such that $(\geqslant_{\mathcal{B}}, >_{\mathcal{B}})$ forms a safe reduction pair with $\mathcal{U}(\mathcal{P}) \subseteq \geqslant_{\mathcal{B}}$ and $\mathcal{P} \subseteq >_{\mathcal{B}}$. If $\mathcal{U}(\mathcal{P}) \subseteq >_{\mathcal{A}}$ then the (innermost) runtime complexity function $\mathsf{rc}^{(\mathsf{i})}_{\mathcal{R}}$ with respect to $\mathcal{R}$ is linear or quadratic, respectively.*

Note that if $\mathcal{U}(\mathcal{P}) = \varnothing$, the compatibility of $\mathcal{U}(\mathcal{P})$ with an SLI is trivially satisfiable. In this special case by taking the SLI $\mathcal{A}$ that interprets all symbols with the zero function, we obtain $\mathsf{dl}(t, \rightarrow) \leqslant \mathsf{G}(t^{\sharp}, >_{\mathcal{A}})$ because $\Delta(\mathcal{A}, \varnothing) = \mathsf{M}_{\mathcal{A}} = 0$. As a consequence of Theorem 31 and Lemma 12 we obtain the following corollary.

Corollary 36. *Let $\mathcal{R}$ be a TRS, let $\mathcal{A}$ be an SLI, let all compound symbols in* $\mathsf{WIDP}(\mathcal{R})$ *be nullary and let $\mathcal{B}$ be a linear or quadratic restricted interpretation such that $(\geqslant_{\mathcal{B}}, >_{\mathcal{B}})$ forms a reduction pair with $\mathcal{U}(\mathsf{DP}(\mathcal{R})) \subseteq \geqslant_{\mathcal{B}}$ and $\mathsf{DP}(\mathcal{R}) \subseteq >_{\mathcal{B}}$. If in addition $\mathcal{U}(\mathsf{DP}(\mathcal{R})) \subseteq >_{\mathcal{A}}$ then the innermost runtime complexity function $\mathsf{rc}^{\mathsf{i}}_{\mathcal{R}}$ with respect to $\mathcal{R}$ is linear or quadratic, respectively.*

Corollary 36 establishes (for the first time) a method to analyse the derivation length induced by the standard dependency pair method for innermost rewriting. More general, if all compound symbols in $\mathsf{WIDP}(\mathcal{R})$ are nullary and there exists a collapsible reduction pair $(\gtrsim, \succ)$ such that $\mathcal{U}(\mathcal{P}) \subseteq \gtrsim$ and $\mathcal{P} \subseteq \succ$, then the innermost runtime complexity of $\mathcal{R}$ is linear in the maximal length of $\succ$-descending steps. Clearly for *string rewriting* (cf. [11]) the compound symbols in $\mathsf{WIDP}(\mathcal{R})$ are always nullary and the conditions works quite well for TRSs, too (see Section 7).

7 Experiments

We implemented a complexity analyser based on syntactical transformations for dependency pairs and usable rules together with polynomial orders (based on [16]). To deal efficiently with polynomial interpretations, the issuing constraints are encoded in *propositional logic* in a similar spirit as in [17]. Assignments are found by employing a state-of-the-art SAT solver, in our case MiniSat[5]. Furthermore, strongly linear interpretations are handled by a decision procedure for Presburger arithmetic. As suitable test bed we used the rewrite systems in the Termination Problem Data Base version 4.0.[6] This test bed comprises 1679 TRSs, including 358 TRSs for innermost rewriting. 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 30 seconds, the times in the tables are given in seconds. Table 1 summarises the results of the conducted experiments.[7] Text written in *italics* below the number of successes or failures indicates total time of success cases or failure cases, respectively.[8]

[5] `http://minisat.se/`.

[6] `http://www.lri.fr/~marche/tpdb/`

[7] For full experimental evidence see `http://www.jaist.ac.jp/~hirokawa/08a/`

[8] Sum of numbers in each column may be less than 1679 because of stack overflow.

Table 1. Experimental Results for TRSs

Linear Runtime Complexities							
	full rewriting				*innermost rewriting*		
	LC	Cor. 18	Cor. 35	both	Cor. 18 (DP)	Cor. 35 (DP)	both
success	139	138	119	161	143 (135)	128 (113)	170
	15	*21*	*18*	*33*	*21* (*20*)	*21* (*15*)	*34*
failure	1529	1501	1560	1478	1495 (1502)	1550 (1565)	1467
	1564	*2517*	*152*	*2612*	*2489* (*2593*)	*180* (*149*)	*2580*
timeout	11	40	0	40	41 (42)	1 (1)	42

Quadratic Runtime Complexities							
	full rewriting				*innermost rewriting*		
	QC	Cor. 18	Cor. 35	both	Cor. 18 (DP)	Cor. 35 (DP)	both
success	176	169	124	188	168 (152)	125 (109)	189
	473	*598*	*254*	*781*	*564* (*457*)	*237* (*128*)	*755*
failure	702	657	1486	601	654 (707)	1486 (1502)	593
	2569	*2591*	*527*	*2700*	*2522* (*2461*)	*602* (*546*)	*2570*
timeout	799	852	69	890	856 (816)	68 (68)	896

We use the following abbreviations: The method LC (QC) refers to compatibility with *linear (quadratic) restricted interpretation*, cf. Section 3. In interpreting defined and dependency pair functions, we restrict the search to polynomials in the range $\{0, 1, \ldots, 5\}$. The upper half of Table 1 shows the experimental results for linear runtime complexities based on LC. The columns marked "Cor. 18" and "Cor. 35" refer to the applicability of the respective corollaries. In the column marked "both" we indicate the results, we obtain when we first try to apply Corollary 35 and if this fails Corollary 18. The lower half summarises experimental results for quadratic runtime complexities based on QC. On the studied test bed there are 1567 TRSs such that one may switch from $\mathsf{WIDP}(\mathcal{R})$ to $\mathsf{DP}(\mathcal{R})$. For the individual tests, we indicated the results in parentheses for this versions of Corollary 18 and Corollary 35.

8 Conclusion

In this paper we studied the runtime complexity of rewrite systems. We have established a variant of the dependency pair method that is applicable in this context and is easily mechanisable. In particular our findings extend the class of TRSs whose *linear* or *quadratic* runtime complexity can be detected automatically. We provided ample numerical data for assessing the viability of the method. To conclude, we mention possible future work. In the experiments presented, we have restricted our attention to interpretation based methods inducing linear or quadratic (innermost) runtime complexity. Recently in [18] a restriction of the multiset path order, called *polynomial path order* has been introduced that induces polynomial runtime complexity. In future work we will test to what extent this is effectively combinable with our Main Theorem. Furthermore, we strive to extend the approach presented here to handle dependency graphs [7].

References

1. 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)
2. Hofbauer, D.: Termination proofs by multiset path orderings imply primitive recursive derivation lengths. TCS 105(1), 129–140 (1992)
3. Weiermann, A.: Termination proofs for term rewriting systems with lexicographic path orderings imply multiply recursive derivation lengths. TCS 139, 355–362 (1995)
4. Hofbauer, D.: Termination proofs by context-dependent interpretations. In: Middeldorp, A. (ed.) RTA 2001. LNCS, vol. 2051, pp. 108–121. Springer, Heidelberg (2001)
5. Moser, G.: Derivational complexity of Knuth Bendix orders revisited. In: Hermann, M., Voronkov, A. (eds.) LPAR 2006. LNCS (LNAI), vol. 4246, pp. 75–89. Springer, Heidelberg (2006)
6. Geser, A., Hofbauer, D., Waldmann, J., Zantema, H.: On tree automata that certify termination of left-linear term rewriting systems. IC 205(4), 512–534 (2007)
7. Arts, T., Giesl, J.: Termination of term rewriting using dependency pairs. TCS 236, 133–178 (2000)
8. Giesl, J., Arts, T., Ohlebusch, E.: Modular termination proofs for rewriting using dependency pairs. JSC 34(1), 21–58 (2002)
9. Hirokawa, N., Middeldorp, A.: Tyrolean termination tool: Techniques and features. IC 205, 474–511 (2007)
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. Giesl, J., Thiemann, R., Schneider-Kamp, P., Falke, S.: Mechanizing and improving dependency pairs. JAR 37(3), 155–203 (2006)
13. Bonfante, G., Cichon, A., Marion, J.Y., Touzet, H.: Algorithms with polynomial interpretation termination proof. JFP 11(1), 33–53 (2001)
14. Geser, A.: Relative Termination. PhD thesis, Universität Passau (1990)
15. Steinbach, J., Kühler, U.: Check your ordering – termination proofs and open problems. Technical Report SR-90-25, Universität Kaiserslautern (1990)
16. Contejean, E., Marché, C., Tomás, A.P., Urbain, X.: Mechanically proving termination using polynomial interpretations. JAR 34(4), 325–363 (2005)
17. 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)
18. 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)

Canonical Inference for Implicational Systems[⋆]

Maria Paola Bonacina[1,⋆⋆] and Nachum Dershowitz[2,⋆⋆⋆]

[1] Dipartimento di Informatica, Università degli Studi di Verona, Italy
mariapaola.bonacina@univr.it
[2] School of Computer Science, Tel Aviv University, Ramat Aviv 69978, Israel
Nachum.Dershowitz@cs.tau.ac.il

Abstract. *Completion* is a general paradigm for applying inferences to generate a canonical presentation of a logical theory, or to semi-decide the validity of theorems, or to answer queries. We investigate what *canonicity* means for *implicational systems* that are axiomatizations of *Moore families* – or, equivalently, of propositional *Horn theories*. We build a correspondence between implicational systems and associative-commutative rewrite systems, give deduction mechanisms for both, and show how their respective inferences correspond. Thus, we exhibit completion procedures designed to generate canonical systems that are "optimal" for forward chaining, to compute minimal models, and to generate canonical systems that are *rewrite-optimal*. Rewrite-optimality is a new notion of "optimality" for implicational systems, one that takes contraction by simplification into account.

1 Introduction

Knowledge compilation is the transformation of a knowledge base into a form that makes efficient reasoning possible (e.g., [17,8,14]). In automated reasoning the knowledge base is often the "presentation" of a theory, where we use "presentation" to mean a set of formulæ, reserving "theory" for a presentation with all its theorems. From the perspective taken here, *canonicity* of a presentation depends on the availability of the best proofs, or *normal-form proofs*. Proofs are measured by *proof orderings*, and the most desirable are the minimal proofs. Since a minimal proof in a certain presentation may not be minimal in a larger presentation, normal-form proofs are the minimal proofs in the largest presentation, that is, in a deductively-closed presentation. However, what is a deductively-closed presentation depends on the choice of *deduction mechanism*. Thus, the choices of normal form and deduction mechanism are intertwined.

An archetypal instance of knowledge compilation is *completion* of *equational theories*, where normal-form proofs are *valley proofs*, that is, proofs where equations only decrease terms: a given presentation E is transformed into an equivalent, *ground-convergent* presentation $E^\sharp$, such that for all theorems $\forall\bar{x}\ u \simeq v$, $E^\sharp$

[⋆] A longer version, "Canonical Ground Horn Theories", is available as RR49/2007, DI, UniVR at http://profs.sci.univr.it/∼bonacina/canonicity.html

[⋆⋆] Research supported in part by MIUR grant no. 2003-097383.

[⋆⋆⋆] Research supported in part by the Israel Science Foundation (grant no. 250/05).

offers a valley proof of $\widetilde{u} \simeq \widetilde{v}$, where $\widetilde{u}$ and $\widetilde{v}$ are u and v with their variables $\bar{x}$ replaced by Skolem constants. Since ground-convergent means *terminating* and *ground-confluent*, if $E^\sharp$ is *finite*, it serves as *decision procedure*, because validity can be decided by "blind" rewriting. If $E^\sharp$ is also *reduced*, it is called *canonical*, and it is unique for the assumed ordering, a property first noticed by Mike Ballantyne (see [12]). Otherwise, completion semi-decides validity by working refutationally on E and $\widetilde{u} \not\simeq \widetilde{v}$ (see, e.g., [11,1,6] for basic definitions and more references).

More generally, the notion of *canonicity* can be articulated into three properties of increasing strength (e.g., [4]): a presentation is *complete* if it affords a normal-form proof for each theorem, it is *saturated*, if it supports all normal-form proofs for all theorems, and *canonical*, if it is both saturated and *contracted*, that is, it contains no redundancies. If minimal proofs are unique, complete and saturated coincide. The general properties "saturated" and "contracted" correspond to convergent and reduced in the equational case.

This paper studies canonicity for implicational systems. An *implicational system* is a set of implications, whose family of models is a *Moore family*, meaning that it is closed under intersection (see [3,2]). A Moore family defines a *closure operator* that associates with any set the least element of the Moore family that includes it. Moore families, closure operators and implicational systems have played a rôle in a variety of fields in computer science, including relational databases, data mining, artificial intelligence, logic programming, lattice theory and abstract interpretations. We refer to [7] and [2] for surveys, including applications, related formalisms and historical notes.

An implicational systems can be regarded as a Horn presentation of its Moore family. Since a Moore family may be presented by different implicational systems, it makes sense to define and generate implicational systems that are "optimal", or "minimal", or "canonical" in a suitable sense, and allow one to compute their associated closure operator efficiently. Bertet and Nebut [3] proposed the notions of *directness* of implicational systems, optimizing computation by forward chaining, and *direct-optimality* of implicational systems, which adds an optimization step based on a symbol count. Bertet and Monjardet [2] considered other candidates and proved them all equal to direct-optimality, which, therefore, earned the appellation *canonical-directness*.

We investigate correspondences between "optimal" implicational systems (direct, direct-optimal) and canonical rewrite systems. This requires us to establish an equivalence between implicational systems and associative-commutative rewrite systems, and to define and compare their respective deduction mechanisms. The rewriting framework allows one to compute the image of a given set according to the closure operator associated with the implicational system, already during saturation of the system. Computing the closure amounts to generating minimal models, which may have practical applications. Comparisons of presentations and inferences are complemented at a deeper level by comparisons of the underlying proof orderings. We observe that direct-optimality can be simulated by normalization with respect to a different proof ordering than the one assumed by rewriting, and this discrepancy leads us to introduce a new notion of *rewrite-optimality.* Thus, while directness corresponds to saturation in an expansion-oriented deduction mechanism, rewrite-optimality corresponds to canonicity.

2 Background

Let V be a vocabulary of propositional variables. For $a \in V$, a and $\neg a$ are positive and negative literals, respectively; a *clause* is a disjunction of literals, that is *positive* (*negative*), if all its literals are, and *unit*, if it is made of a single literal. A *Horn clause* has at most one positive literal, so positive unit clauses and purely negative clauses are special cases. A *Horn presentation* is a set of non-negative Horn clauses. It is customary to write a Horn clause $\neg a_1 \vee \cdots \vee \neg a_n \vee c$, $n \geq 0$, as the implication or *rule* $a_1 \cdots a_n \Rightarrow c$. A Horn clause is *trivial* if the *conclusion* c is the same as one of the *premises* a_i.

An *implicational system* S is a binary relation $S \subseteq \mathcal{P}(V) \times \mathcal{P}(V)$, read as a set of implications $a_1 \cdots a_n \Rightarrow c_1 \cdots c_m$, for $a_i, c_j \in V$, with both sides understood as conjunctions of distinct propositions (see, e.g., [3,2]). Using upper case Latin letters for sets, such an implication is written $A \Rightarrow_S B$ to specify that $A \Rightarrow B \in S$. If all right-hand sides are singletons, S is a *unary implicational system.* Clearly, any non-negative Horn clause is such a unary implication and vice-versa, and any non-unary implication can be decomposed into m unary implications, one for each c_i.

Since an implication $a_1 \cdots a_n \Rightarrow c_1 \cdots c_m$ is equivalent to the bi-implication $a_1 \cdots a_n c_1 \cdots c_m \Leftrightarrow a_1 \cdots a_n$, again with both sides understood as conjunctions, it can also be translated into a rewrite rule $a_1 \cdots a_n c_1 \cdots c_m \ \rightarrow \ a_1 \cdots a_n$, where juxtaposition stands for associative-commutative-idempotent conjunction, and the arrow $\rightarrow$ signifies logical *equivalence* (see, e.g., [9,5]). A positive literal c is translated into a rule $c \rightarrow \mathit{true}$, where *true* is a special constant. We will be making use of a well-founded ordering $\succ$ on $V \cup \{\mathit{true}\}$, wherein *true* is minimal. Conjunctions of propositions are compared by the multiset extension of $\succ$, also denoted $\succ$, so that $a_1 \ldots a_n c_1 \ldots c_m \succ a_1 \ldots a_n$. A rewrite rule $P \rightarrow Q$ is measured by the multiset $\{\!\{P, Q\}\!\}$, and these measures are ordered by a second multiset extension of $\succ$, written $\succ_L$ to avoid confusion. Sets of rewrite rules are measured by multisets of multisets (e.g., containing $\{\!\{P, Q\}\!\}$ for each $P \rightarrow Q$ in the set) compared by the multiset extension of $\succ_L$, denoted $\succ_C$.

A subset $X \subseteq V$ represents the propositional interpretation that assigns the value *true* to all elements in X and *false* to all others. Accordingly, X is said to *satisfy* an implication $A \Rightarrow B$ if either $B \subseteq X$ or else $A \not\subseteq X$. Similarly, we say that X *satisfies* an implicational system S, or is a *model* of S, denoted $X \models S$, if X satisfies all implications in S.

A *Moore family* on V is a family $\mathcal{F}$ of subsets of V that contains V and is closed under intersection. Moore families are in one-to-one correspondence with closure operators, where a *closure operator* on V is an operator $\varphi\colon \mathcal{P}(V) \rightarrow \mathcal{P}(V)$ that is: (i) *monotone*: $X \subseteq X'$ implies $\varphi(X) \subseteq \varphi(X')$; (ii) *extensive*: $X \subseteq \varphi(X)$;

and (iii) *idempotent*: $\varphi(\varphi(X)) = \varphi(X)$. The Moore family $\mathcal{F}_\varphi$ associated with a given closure operator φ is the set of all fixed points of φ:

$$\mathcal{F}_\varphi \quad \overset{!}{=} \quad \{X \subseteq V : X = \varphi(X)\} ,$$

using $\overset{!}{=}$ to signify definitions. The closure operator $\varphi_\mathcal{F}$ associated with a given Moore family $\mathcal{F}$ maps any $X \subseteq V$ to the least element of $\mathcal{F}$ that contains X:

$$\varphi_\mathcal{F}(X) \quad \overset{!}{=} \quad \bigcap\{Y \in \mathcal{F} : X \subseteq Y\} .$$

The Moore family $\mathcal{F}_S$ associated with a given implicational system S is the family of the *propositional models* of S, in the sense given above:

$$\mathcal{F}_S \quad \overset{!}{=} \quad \{X \subseteq V : X \models S\} .$$

Two implicational systems S and S' that have the same Moore family, $\mathcal{F}_S = \mathcal{F}_{S'}$, are said to be *equivalent*. Every Moore family $\mathcal{F}$ can be presented at least by one implicational system, for instance $\{X \Rightarrow \varphi_\mathcal{F}(X) : X \subseteq V\}$.

Combining the notions of closure operator for a Moore family, and Moore family associated with an implicational system, the closure operator φ_S for implicational system S maps any $X \subseteq V$ to the least model of S that satisfies X [3]:

$$\varphi_S(X) \quad \overset{!}{=} \quad \bigcap\{Y \subseteq V : Y \supseteq X \wedge Y \models S\} .$$

Example 1. Let S be $\{a \Rightarrow b,\ ac \Rightarrow d,\ e \Rightarrow a\}$. The Moore family $\mathcal{F}_S$ is $\{\emptyset, b, c, d, ab, bc, bd, cd, abd, abe, bcd, abcd, abde, abcde\}$, and $\varphi_S(ae) = abe$. □

The obvious syntactic correspondence between Horn presentations and implicational systems is matched by a semantic correspondence between Horn theories and Moore families, since Horn theories are those theories whose models are closed under intersection, a fact observed first by McKinsey [16] (see also [15]).

A *(one-step) deduction mechanism* $\rightsquigarrow$ is a binary relation over presentations. A deduction step $Q \rightsquigarrow Q \cup Q'$ is an *expansion* provided $Q' \subseteq \mathit{Th}\,Q$, where $\mathit{Th}\,Q$ is the set of theorems of Q. A deduction step $Q \cup Q' \rightsquigarrow Q$ is a *contraction* provided $Q \cup Q' \gtrsim Q$, which means $\mathit{Th}\,(Q \cup Q') = \mathit{Th}\,Q$, and for all theorems, Q offers a proof that is smaller or equal than that in $Q \cup Q'$ in a well-founded proof ordering. A sequence $Q_0 \rightsquigarrow Q_1 \rightsquigarrow \cdots$ is a *derivation*, whose result, or *limit*, is the set of *persistent* formulæ: $Q_\infty \overset{!}{=} \cup_j \cap_{i \geq j} Q_i$. A fundamental requirement of derivations is *fairness*, doing all expansion inferences that are needed to achieve the desired degree of proof normalization: a *fair* derivation generates a complete limit and a *uniformly fair* derivation generates a saturated limit. For canonicity, systematic application of contraction is also required: a *contracting* derivation generates a contracted limit. In this paper we assume that minimal proofs are unique, which is reasonable for propositional Horn theories, so that complete and saturated coincide, as do fair and uniformly fair. If minimal proofs are not unique, all our results still hold, provided the hypothesis of fairness of derivations is replaced by uniform fairness. We refer to [4] for formal definitions of these notions.

3 Direct Systems

A *direct* implicational system allows one to compute $\varphi_S(X)$ in one single round of *forward chaining*:

Definition 1 (Directness [3, Def. 1]). *An implicational system S is* direct *if $\varphi_S(X) = S(X)$, where $S(X) \overset{!}{=} X \cup \bigcup\{B : A \Rightarrow_S B \wedge A \subseteq X\}$.*

In general, $\varphi_S(X) = S^*(X)$, where

$$S^0(X) = X, \qquad S^{i+1}(X) = S(S^i(X)), \qquad S^*(X) = \bigcup_i S^i(X).$$

Since S, X and V are all finite, $S^*(X) = S^k(X)$ for the smallest k such that $S^{k+1}(X) = S^k(X)$.

Example 2. The implicational system $S = \{ac \Rightarrow d, e \Rightarrow a\}$ is not direct. Indeed, for $X = ce$, the computation of $\varphi_S(X) = \{acde\}$ requires two rounds of forward chaining, because only after a has been added by $e \Rightarrow a$, can d be added by $ac \Rightarrow d$. That is, $S(X) = \{ace\}$ and $\varphi_S(X) = S^2(X) = S^*(X) = \{acde\}$. □

Generalizing this example, it is sufficient to have two implications $A \Rightarrow_S B$ and $C \Rightarrow_S D$ such that $A \subseteq X$, $C \not\subseteq X$ and $C \subseteq X \cup B$, for $\varphi_S(X)$ to require more than one iteration of forward chaining. Since $A \subseteq X$, but $C \not\subseteq X$, the first round adds B, but not D; since $C \subseteq X \cup B$, D is added in a second round. In the above example, $A \Rightarrow B$ is $e \Rightarrow a$ and $C \Rightarrow D$ is $ac \Rightarrow d$. The conditions $A \subseteq X$ and $C \subseteq X \cup B$ are equivalent to $A \cup (C \setminus B) \subseteq X$, because $C \subseteq X \cup B$ means that whatever is in C and not in B must be in X. Thus, to collapse the two iterations of forward chaining into one, it is sufficient to add the implication $A \cup (C \setminus B) \Rightarrow_S D$. In the example $A \cup (C \setminus B) \Rightarrow_S D$ is $ce \Rightarrow d$. This mechanism can be defined in more abstract terms as the following inference rule:

Implicational overlap

$$\frac{A \Rightarrow BO \quad CO \Rightarrow D}{AC \Rightarrow D} \quad B \cap C = \emptyset \neq O$$

One inference step of this rule will be denoted $\vdash_I$. The condition $O \neq \emptyset$ is included, because otherwise $AC \Rightarrow D$ is subsumed by $C \Rightarrow D$. Also, if $B \cap C$ is not empty, then an alternate inference is more general. Thus, directness can be characterized as follows:

Definition 2 (Generated direct system [3, Def. 4]). *Given an implicational system S, the* direct *implicational system $I(S)$ generated from S is the smallest implicational system containing S and closed with respect to implicational overlap.*

A main theorem of [3] shows that indeed $\varphi_S(X) = I(S)(X)$. Let $\rightsquigarrow_I$ be the deduction mechanism that generates and adds implications by implicational overlap: clearly, $\rightsquigarrow_I$ steps are expansion steps. Thus, we have:

Proposition 1. *Given an implicational system S, for all fair derivations $S = S_0 \rightsquigarrow_I S_1 \rightsquigarrow_I \cdots$, $S_\infty = I(S)$.*

Proof. By fairness, S_∞ is saturated, and therefore closed with respect to implicational overlap. Since $\rightsquigarrow_I$ deletes nothing, S_∞ contains S. Since $\rightsquigarrow_I$ adds nothing beside implicational overlaps, S_∞ is equal to the smallest system with these properties, that is, $S_\infty = I(S)$.

By applying the translation of implications into rewrite rules (cf. Section 2), we define:

Definition 3 (Associated rewrite system). *Given $X \subseteq V$, its* associated rewrite system *is $R_X \overset{!}{=} \{x \to true : x \in X\}$. For an implicational system S, its* associated rewrite system *is $R_S \overset{!}{=} \{AB \to A : A \Rightarrow_S B\}$. Given S and X we can also form the rewrite system $R_X^S \overset{!}{=} R_X \cup R_S$.*

Example 3. If $S = \{a \Rightarrow b, ac \Rightarrow d, e \Rightarrow a\}$, then $R_S = \{ab \to a, acd \to ac, ae \to e\}$. If $X = ae$, then $R_X = \{a \to true, e \to true\}$. Thus, $R_X^S = \{a \to true, e \to true, ab \to a, acd \to ac, ae \to e\}$. □

We show that there is a correspondence between implicational overlap and the classical notion of overlap between monomials in Boolean rewriting (e.g., [5]):

Equational overlap

$$\frac{AO \to B \quad CO \to D}{M \to N} \quad A \cap C = \emptyset \neq O,\ M \succ N$$

where M and N are the normal-forms of BC and AD with respect to $\{AO \to B, CO \to D\}$. One inference step of this rule will be denoted $\vdash_E$.

Equational overlap combines expansion, the generation of $BC \leftrightarrow AD$, with contraction – its normalization to $M \to N$. This sort of contraction applied to normalize a newly generated formula, is called *forward contraction*. The contraction applied to reduce an already established equation is called *backward contraction*. Let $\rightsquigarrow_E$ be the deduction mechanism of equational overlap: then, $\rightsquigarrow_E$ features expansion and forward contraction.

Example 4. For $S = \{ac \Rightarrow d, e \Rightarrow a\}$ as in Example 2, we have $R_S = \{acd \to ac, ae \to e\}$, and the overlap of the two rewrite rules gives $ace \leftarrow acde \to cde$. Since $ace \to ce$, equational overlap yields the rewrite rule $cde \to ce$, which corresponds to the implication $ce \Rightarrow d$ generated by implicational overlap. □

Without loss of generality, from now on we consider only systems made of unary implications or Horn clauses (cf. Section 2). Since it is designed to produce a direct system, implicational overlap "unfolds" the forward chaining in the implicational system. Since forward chaining is complete for Horn logic, it is coherent to expect that the only non-trivial equational overlaps are those corresponding to implicational overlaps:

Lemma 1. *If $A \Rightarrow B$ and $C \Rightarrow D$ are two non-trivial Horn clauses* ($|B| = |D| = 1$, $B \not\subseteq A$, $D \not\subseteq C$), *then if $A \Rightarrow B, C \Rightarrow D \vdash_I E \Rightarrow D$ by implicational overlap, then $AB \rightarrow A, CD \rightarrow C \vdash_E DE \rightarrow E$ by equational overlap, and vice-versa. Furthermore, all other equational overlaps are trivial.*

Proof. (If direction.) Assume $A \Rightarrow B, C \Rightarrow D \vdash_I E \Rightarrow D$. Since B is a singleton by hypothesis, it must be that the consequent of the first implication and the antecedent of the second one overlap on B. Thus, $C \Rightarrow D$ is $BF \Rightarrow D$ and the implicational overlap of $A \Rightarrow B$ and $BF \Rightarrow D$ generates $AF \Rightarrow D$. The corresponding rewrite rules are $AB \rightarrow A$ and $BFD \rightarrow BF$, which also overlap on B yielding the equational overlap

$$AFD \leftarrow ABFD \rightarrow ABF \rightarrow AF \ ,$$

which generates the corresponding rule $AFD \rightarrow AF$.
(Only if direction.) If $AB \rightarrow A, CD \rightarrow C \vdash_E DE \rightarrow E$, the rewrite rules $AB \rightarrow A$ and $CD \rightarrow C$ can overlap in four ways: $B \cap C \neq \emptyset$, $A \cap D \neq \emptyset$, $A \cap C \neq \emptyset$ and $B \cap D \neq \emptyset$, which we consider in order.

1. $B \cap C \neq \emptyset$: Since B is a singleton, it must be $B \cap C = B$, hence $C = BF$ for some F. Thus, $CD \rightarrow C$ is $BFD \rightarrow BF$, and the overlap of $AB \rightarrow A$ and $BFD \rightarrow BF$ is the same as above, yielding $AFD \rightarrow AF$. The corresponding implications $A \Rightarrow B$ and $BF \Rightarrow D$ generate $AF \Rightarrow D$ by implicational overlap.
2. $A \cap D \neq \emptyset$: This case is symmetric to the previous one.
3. $A \cap C \neq \emptyset$: Let $A = FO$ and $C = OG$, so that the rules are $FOB \rightarrow FO$ and $OGD \rightarrow OG$, with $O \neq \emptyset$ and $F \cap G = \emptyset$. The resulting equational overlap is trivial: $FOG \leftarrow FOGD \leftarrow FBOGD \rightarrow FBOG \rightarrow FOG$.
4. $B \cap D \neq \emptyset$: Since B and D are singletons, it must be $B \cap D = B = D$, and rules $AB \rightarrow A$ and $CB \rightarrow C$ produce the trivial overlap $AC \leftarrow ABC \rightarrow AC$.

□

Lemma 1 yields the following correspondence between deduction mechanisms:

Lemma 2. *For all implicational systems S, $S \leadsto_I S'$ if and only if $R_S \leadsto_E R_{S'}$.*

Proof. If $S \leadsto_I S'$ then $R_S \leadsto_E R_{S'}$ follows from the if direction of Lemma 1. If $R_S \leadsto_E R'$ then $S \leadsto_I S'$ and $R' = R_{S'}$ follows from the only-if direction of Lemma 1. □

The next theorem shows that for fair derivations the process of completing S with respect to implicational overlap, and turning the result into a rewrite system, commutes with the process of translating S into the rewrite system R_S, and then completing it with respect to equational overlap.

Theorem 1. *For every implicational system S, and for all fair derivations $S = S_0 \leadsto_I S_1 \leadsto_I \cdots$ and $R_S = R_0 \leadsto_E R_1 \leadsto_E \cdots$, we have*

$$R_{(S_\infty)} = (R_S)_\infty \ .$$

Proof.

(a) $R_{(S_\infty)} \subseteq (R_S)_\infty$: for any $AB \rightarrow A \in R_{(S_\infty)}$, $A \Rightarrow B \in S_\infty$ by Definition 3; then $A \Rightarrow B \in S_j$ for some $j \geq 0$. Let j be the smallest such index. If $j = 0$, or $S_j = S$, $AB \rightarrow A \in R_S$ by Definition 3, and $AB \rightarrow A \in (R_S)_\infty$, because $\rightsquigarrow_E$ features no backward contraction. If $j > 0$, $A \Rightarrow B$ is generated at stage j by implicational overlap. By Lemma 2 and by fairness of $R_0 \rightsquigarrow_E R_1 \rightsquigarrow_E \cdots$, $AB \rightarrow A \in R_k$ for some $k > 0$. Then $AB \rightarrow A \in (R_S)_\infty$, since $\rightsquigarrow_E$ features no backward contraction.

(b) $(R_S)_\infty \subseteq R_{(S_\infty)}$: for any $AB \rightarrow A \in (R_S)_\infty$, $AB \rightarrow A \in R_j$ for some $j \geq 0$. Let j be the smallest such index. If $j = 0$, or $R_j = R_S$, $A \Rightarrow B \in S$ by Definition 3, and $A \Rightarrow B \in S_\infty$, because $\rightsquigarrow_I$ features no backward contraction. Hence $AB \rightarrow A \in R_{(S_\infty)}$. If $j > 0$, $AB \rightarrow A$ is generated at stage j by equational overlap. By Lemma 2 and by fairness of $S_0 \rightsquigarrow_I S_1 \rightsquigarrow_I \cdots$, $A \Rightarrow B \in S_k$ for some $k > 0$. Then $A \Rightarrow B \in S_\infty$, since $\rightsquigarrow_I$ features no backward contraction, and $AB \rightarrow A \in R_{(S_\infty)}$ by Definition 3. □

Since the limit of a fair $\rightsquigarrow_I$-derivation is $I(S)$, it follows that:

Corollary 1. *For every implicational system S, and for all fair derivations $S = S_0 \rightsquigarrow_I S_1 \rightsquigarrow_I \cdots$ and $R_S = R_0 \rightsquigarrow_E R_1 \rightsquigarrow_E \cdots$, we have*

$$R_{(I(S))} = (R_S)_\infty \,.$$

4 Computing Minimal Models

The motivation for generating $I(S)$ from S is to be able to compute, for any subset $X \subseteq V$, its minimal S-model $\varphi_S(X)$ in one round of forward chaining. In other words, one envisions a two-stage process: in the first stage, S is saturated with respect to implicational overlap to generate $I(S)$; in the second stage, forward chaining is applied to $I(S) \cup X$ to generate $\varphi_{I(S)}(X) = \varphi_S(X)$. These two stages can be replaced by one: for any $X \subseteq V$ we can compute $\varphi_S(X) = \varphi_{I(S)}(X)$, by giving the rewrite system R_X^S as input to completion and extracting rules of the form $x \rightarrow \mathit{true}$. For this purpose, the deduction mechanism is enriched with contraction rules, for which we employ a double inference line:

Simplification

$$\frac{AC \rightarrow B \quad C \rightarrow D}{\overline{AD \rightarrow B \quad C \rightarrow D}}\; AD \succ B \qquad \frac{AC \rightarrow B \quad C \rightarrow D}{\overline{B \rightarrow AD \quad C \rightarrow D}}\; B \succ AD$$

$$\frac{B \rightarrow AC \quad C \rightarrow D}{\overline{B \rightarrow AD \quad C \rightarrow D}}\,,$$

where A can be empty, and

Deletion

$$\frac{A \leftrightarrow A}{} \,,$$

which eliminates trivial equivalences.

Let $\leadsto_R$ denote the deduction mechanism that extends $\leadsto_E$ with simplification and deletion. Thus, in addition to the simplification applied as forward contraction within equational overlap, there is simplification applied as backward contraction to any rule. The following theorem shows that the completion of R_X^S with respect to $\leadsto_R$ generates a limit that includes the least S-model of X:

Theorem 2. *For all $X \subseteq V$, implicational systems S, and fair derivations $R_X^S = R_0 \leadsto_R R_1 \leadsto_R \cdots$, if $Y = \varphi_S(X) = \varphi_{I(S)}(X)$, then*

$$R_Y \subseteq (R_X^S)_\infty \,.$$

Proof. By Definition 3, $R_Y = \{x \to \mathit{true} \colon x \in Y\}$. The proof is by induction on the construction of $Y = \varphi_S(X)$.
Base case: If $x \in Y$ because $x \in X$, then $x \to \mathit{true} \in R_X$, $x \to \mathit{true} \in R_X^S$ and $x \to \mathit{true} \in (R_X^S)_\infty$, since a rule in the form $x \to \mathit{true}$ is persistent.
Inductive case: If $x \in Y$ because for some $A \Rightarrow_S B$, $B = x$ and $A \subseteq Y$, then $AB \to A \in R_S$ and $AB \to A \in R_X^S$. By the induction hypothesis, $A \subseteq Y$ implies that, for all $z \in A$, $z \in Y$ and $z \to \mathit{true} \in (R_X^S)_\infty$. Let $j > 0$ be the smallest index in the derivation $R_0 \leadsto_E R_1 \leadsto_E \cdots$ such that for all $z \in A$, $z \to \mathit{true} \in R_j$. Then there is an $i > j$ such that $x \to \mathit{true} \in R_i$, because the rules $z \to \mathit{true}$ simplify $AB \to A$ to $x \to \mathit{true}$. It follows that $x \to \mathit{true} \in (R_X^S)_\infty$, since a rule in the form $x \to \mathit{true}$ is persistent. □

Then, the least S-model of X can be extracted from the saturated set:

Corollary 2. *For all $X \subseteq V$, implicational systems S, and fair derivations $R_X^S = R_0 \leadsto_R R_1 \leadsto_R \cdots$, if $Y = \varphi_S(X) = \varphi_{I(S)}(X)$, then*

$$R_Y = \{x \to \mathit{true} \colon x \to \mathit{true} \in (R_X^S)_\infty\} \,.$$

Proof. If $x \to \mathit{true} \in (R_X^S)_\infty$, then $x \in R_Y$ by the soundness of equational overlap and simplification. The other direction was established in Theorem 2.
□

Example 5. Let $S = \{ac \Rightarrow d, e \Rightarrow a, bd \Rightarrow f\}$ and $X = ce$. Then $Y = \varphi_S(X) = acde$, and $R_Y = \{a \to \mathit{true}, c \to \mathit{true}, d \to \mathit{true}, e \to \mathit{true}\}$. On the other hand, for $R_S = \{acd \to ac, ae \to e, bdf \to bd\}$ and $R_X = \{c \to \mathit{true}, e \to \mathit{true}\}$, completion gives $\left(R_X^S\right)_\infty = \{c \to \mathit{true}, e \to \mathit{true}, a \to \mathit{true}, d \to \mathit{true}, bf \to b\}$, where $a \to \mathit{true}$ is generated by simplification of $ae \to e$ with respect to $e \to \mathit{true}$, $d \to \mathit{true}$ is generated by simplification of $acd \to ac$ with respect to $c \to \mathit{true}$ and $a \to \mathit{true}$, and $bf \to b$ is generated by simplification of $bdf \to bd$ with respect to $d \to \mathit{true}$. So, $\left(R_X^S\right)_\infty$ includes R_Y, which is made exactly of the rules in the form

$x \to \mathit{true}$ of $\left(R_X^S\right)_\infty$. The direct system $I(S)$ contains the implication $ce \Rightarrow d$, generated by implicational overlap from $ac \Rightarrow d$ and $e \Rightarrow a$. The corresponding equational overlap of $acd \to ac$ and $ae \to e$ gives $ce \leftarrow ace \leftarrow acde \to cde$ and hence generates the rule $cde \to ce$. However, this rule is redundant in the presence of $\{c \to \mathit{true}, e \to \mathit{true}, d \to \mathit{true}\}$ and simplification. □

5 Direct-Optimal Systems

Direct-optimality is defined by adding to directness a requirement of optimality, with respect to a measure $|\, S \,|$ that counts the sum of the number of occurrences of symbols on each of the two sides of each implication in a system S:

Definition 4 (Optimality [3, Sect. 2]). *An implicational system S is* optimal *if, for all equivalent implicational system S', $|\, S \,| \leq |\, S' \,|$ where*

$$|\, S \,| \quad \overset{!}{=} \quad \sum_{A \Rightarrow_S B} |A| + |B| \,,$$

where $|A|$ is the cardinality of set A.

From an implicational system S, one can generate an equivalent implicational system that is both direct and optimal, denoted $D(S)$, with the following necessary and sufficient properties (cf. [3, Thm. 2]):

- *extensiveness*: for all $A \Rightarrow_{D(S)} B$, $A \cap B = \emptyset$;
- *isotony*: for all $A \Rightarrow_{D(S)} B$ and $C \Rightarrow_{D(S)} D$, if $C \subset A$, then $B \cap D = \emptyset$;
- *premise*: for all $A \Rightarrow_{D(S)} B$ and $A \Rightarrow_{D(S)} B'$, $B = B'$;
- *non-empty conclusion*: for all $A \Rightarrow_{D(S)} B$, $B \neq \emptyset$.

This leads to the following characterization:

Definition 5 (Direct-optimal system [3, Def. 5]). *Given a direct system S, the* direct-optimal *system $D(S)$ generated from S contains precisely the implications*

$$A \Rightarrow \bigcup\{B\colon A \Rightarrow_S B\} \setminus \{C\colon D \Rightarrow_S C \wedge D \subset A\} \setminus A \,,$$

for each set A of propositions – provided the conclusion is non-empty.

From the above four properties, we can define an *optimization* procedure, applying – in order – the following rules:

Premise

$$\frac{A \Rightarrow B,\ A \Rightarrow C}{A \Rightarrow BC} \,,$$

Isotony

$$\frac{A \Rightarrow B,\ AD \Rightarrow BE}{A \Rightarrow B,\ AD \Rightarrow E} \,,$$

Extensiveness

$$\frac{AC \Rightarrow BC}{\overline{AC \Rightarrow B}}\,,$$

Definiteness

$$\frac{A \Rightarrow \emptyset}{\overline{}}\,.$$

The first rule merges all rules with the same antecedent A into one and implements the *premise* property. The second rule removes from the consequent thus generated those subsets B that are already implied by subsets A of AD, to enforce *isotony*. The third rule makes sure that antecedents C do not themselves appear in the consequent to enforce *extensiveness*. Finally, implications with empty consequent are eliminated. This latter rule is called *definiteness*, because it eliminates negative clauses, which, for Horn theories, represent queries and are not "definite" (i.e., non-negative) clauses. Clearly, the changes wrought by the optimization rules do not affect the theory. Application of this optimization to the direct implicational system $I(S)$ yields the direct-optimal system $D(S)$ of S.

The following example shows that this notion of optimization does *not* correspond to elimination of redundancies by contraction in completion:

Example 6. Let $S = \{a \Rightarrow b, ac \Rightarrow d, e \Rightarrow a\}$. Then, $I(S) = \{a \Rightarrow b, ac \Rightarrow d, e \Rightarrow a, e \Rightarrow b, ce \Rightarrow d\}$, where $e \Rightarrow b$ is generated by implicational overlap of $e \Rightarrow a$ and $a \Rightarrow b$, and $ce \Rightarrow d$ is generated by implicational overlap of $e \Rightarrow a$ and $ac \Rightarrow d$. Next, optimization replaces $e \Rightarrow a$ and $e \Rightarrow b$ by $e \Rightarrow ab$, so that $D(S) = \{a \Rightarrow b, ac \Rightarrow d, e \Rightarrow ab, ce \Rightarrow d\}$. If we consider the rewriting side, we have $R_S = \{ab \to a, acd \to ac, ae \to e\}$. Equational overlap of $ae \to e$ and $ab \to a$ generates $be \to e$, and equational overlap of $ae \to e$ and $acd \to ac$ generates $cde \to ce$, corresponding to the two implicational overlaps. Thus, $(R_S)_\infty = \{ab \to a, acd \to ac, ae \to e, be \to e, cde \to ce\}$. The rule corresponding to $e \Rightarrow ab$, namely $abe \to e$, would be redundant if added to $(R_S)_\infty$, because it would be reduced to a trivial equivalence by $ae \to e$ and $be \to e$. Thus, the optimization consisting of replacing $e \Rightarrow a$ and $e \Rightarrow b$ by $e \Rightarrow ab$ does not correspond to a rewriting inference. $\square$

The reason for this discrepancy is the different choice of ordering. Seeking direct-optimality means optimizing the overall size of the system. For Example 6, we have $|\{e \Rightarrow ab\}| = 3 < 4 = |\{e \Rightarrow a, e \Rightarrow b\}|$. The corresponding proof ordering measures a proof of a from a set X and an implicational system S by a multiset of pairs $\langle |B|, \#_B S\rangle$, for each $B \Rightarrow_S aC$ such that $B \subseteq X$, where $\#_B S$ is the number of implications in S with antecedent B. A proof of a from $X = \{e\}$ and $\{e \Rightarrow ab\}$ will have measure $\{\!\{\langle 1, 1\rangle\}\!\}$, which is smaller than the measure $\{\!\{\langle 1, 2\rangle, \langle 1, 2\rangle\}\!\}$ of a proof of a from $X = \{e\}$ and $\{e \Rightarrow a, e \Rightarrow b\}$.

Completion, on the other hand, optimizes with respect to the ordering $\succ$. For $\{abe \to e\}$ and $\{ae \to e, be \to e\}$, we have $ae \prec abe$ and $be \prec abe$, so $\{\!\{ae, e\}\!\} \prec_L \{\!\{abe, e\}\!\}$ and $\{\!\{be, e\}\!\} \prec_L \{\!\{abe, e\}\!\}$ in the multiset extension $\succ_L$ of $\succ$, and $\{\!\{\{\!\{ae, e\}\!\}, \{\!\{be, e\}\!\}\}\!\} \prec_C \{\!\{\{\!\{abe, e\}\!\}\}\!\}$ in the multiset extension $\succ_C$ of $\succ_L$.

Indeed, from a rewriting point of view, it is better to have $\{ae \to e, be \to e\}$ than $\{abe \to e\}$, since rules with smaller left hand side are more applicable.

6 Rewrite-Optimality

It is apparent that the differences between direct-optimality and completion arise because of the application of the *premise* rule. Accordingly, we propose an alternative definition of optimality, one that does not require the *premise* property, because symbols in repeated antecedents are counted only once:

Definition 6 (Rewrite-optimality). *An implicational system S is* rewrite-optimal *if $\| S \| \leq \| S' \|$ for all equivalent implicational system S', where the measure $\| S \|$ is defined by:*

$$\| S \| \quad \overset{!}{=} \quad |Ante(S)| + |Cons(S)| \,,$$

for $Ante(S) \overset{!}{=} \{c\colon c \in A,\ A \Rightarrow_S B\}$, the set *of symbols occurring in antecedents, and $Cons(S) \overset{!}{=} \{\!\{c\colon c \in B,\ A \Rightarrow_S B\}\!\}$, the* multiset *of symbols occurring in consequents.*

Unlike Definition 4, where antecedents and consequents contribute equally, here symbols in antecedents are counted only once, because $Ante(S)$ is a set, while symbols in consequents are counted as many times as they appear, since $Cons(S)$ is a multiset. Rewrite-optimality appears to be appropriate for Horn clauses, because the *premise* property conflicts with the decomposition of non-unary implications into Horn clauses. Indeed, if S is a non-unary implicational system, and S_H is the equivalent Horn system obtained by decomposing non-unary implications, the application of the *premise* rule to S_H undoes the decomposition.

Example 7. Applying rewrite optimality to $S = \{a \Rightarrow b, ac \Rightarrow d, e \Rightarrow a\}$ of Example 6, we have $\|\{e \Rightarrow ab\}\| = 3 = \|\{e \Rightarrow a, e \Rightarrow b\}\|$, so that replacing $\{e \Rightarrow a, e \Rightarrow b\}$ by $\{e \Rightarrow ab\}$ is no longer justified. Thus, $D(S) = I(S) = \{a \Rightarrow b, ac \Rightarrow d, e \Rightarrow a, e \Rightarrow b, ce \Rightarrow d\}$, and the rewrite system associated with $D(S)$ is $\{ab \to a, acd \to ac, ae \to e, be \to e, cde \to ce\} = (R_S)_\infty$. A proof ordering corresponding to rewrite optimality would measure a proof of a from a set X and an implicational system S by the set of the cardinalities $|B|$, for each $B \Rightarrow_S aC$ such that $B \subseteq X$. Accordingly, a proof of a from $X = \{e\}$ and $\{e \Rightarrow ab\}$ will have measure $\{\!\{1\}\!\}$, which is the same as the measure of a proof of a from $X = \{e\}$ and $\{e \Rightarrow a, e \Rightarrow b\}$. □

Thus, we deem *canonical* the result of optimization without *premise* rule:

Definition 7 (Canonical system). *Given an implicational system S, the* canonical *implicational system $O(S)$ generated from S is the closure of S with respect to implicational overlap, isotony, extensiveness and definiteness.*

Let $\rightsquigarrow_O$ denote the deduction mechanism that features implicational overlap as expansion rule and the optimization rules except *premise*, namely *isotony*, *extensiveness* and *definiteness*, as contraction rules. Then, we have:

Proposition 2. *Given an implicational system S, for all fair and contracting derivations $S = S_0 \rightsquigarrow_O S_1 \rightsquigarrow_O \cdots$, $S_\infty = O(S)$.*

Proof. If the derivation is fair and contracting, both expansion and contraction rules are applied systematically, hence the result.

The following lemma shows that every inference by $\rightsquigarrow_O$ is covered by an inference in $\rightsquigarrow_R$:

Lemma 3. *For all implicational systems S, if $S \rightsquigarrow_O S'$, then $R_S \rightsquigarrow_R R_{S'}$.*

Proof. We consider four cases, corresponding to the four inference rules in $\rightsquigarrow_O$:

1. *Implicational overlap:* If $S \rightsquigarrow_O S'$ by an implicational overlap step, then $R_S \rightsquigarrow_R R_{S'}$ by equational overlap, by Lemma 2.
2. *Isotony:* For an application of this rule, $S = S'' \cup \{A \Rightarrow B, AD \Rightarrow BE\}$ and $S' = S'' \cup \{A \Rightarrow B, AD \Rightarrow E\}$. Then, $R_S = R_{S''} \cup \{AB \to A, ADBE \to AD\}$. Simplification applies to R_S using $AB \to A$ to rewrite $ADBE \to AD$ to $ADE \to AD$, yielding $R_{S''} \cup \{AB \to A, ADE \to AD\} = R_{S'}$.
3. *Extensiveness:* When this rule applies, $S = S'' \cup \{AC \Rightarrow BC\}$ and $S' = S'' \cup \{AC \Rightarrow B\}$. Then, $R_S = R_{S''} \cup \{ACBC \to AC\}$. By mere idempotence of juxtaposition, $R_S = R_{S''} \cup \{ABC \to AC\} = R_{S'}$.
4. *Definiteness:* If $S = S' \cup \{A \Rightarrow \emptyset\}$, then $R_S = R_{S'} \cup \{A \leftrightarrow A\}$ and an application of deletion eliminates the trivial equation, yielding $R_{S'}$. □

However, the other direction of this lemma does not hold, because $\rightsquigarrow_R$ features simplifications that do not correspond to inferences in $\rightsquigarrow_O$:

Example 8. Assume that the implicational system S includes $\{de \Rightarrow b, b \Rightarrow d\}$. Accordingly, R_S contains $\{deb \to de, bd \to b\}$. A simplification inference applies $bd \to b$ to reduce $deb \to de$ to $be \leftrightarrow de$, which is oriented into $be \to de$, if $b \succ d$, and into $de \to be$, if $d \succ b$. (Were $\rightsquigarrow_R$ equipped with a cancellation inference rule, $be \leftrightarrow de$ could be rewritten to $b \leftrightarrow d$, whence $b \to d$ or $d \to b$.) The deduction mechanism $\rightsquigarrow_O$ can apply implicational overlap to $de \Rightarrow b$ and $b \Rightarrow d$ to generate $de \Rightarrow d$. However, $de \Rightarrow d$ is reduced to $de \Rightarrow \emptyset$ by the extensiveness rule, and $de \Rightarrow \emptyset$ is deleted by the definiteness rule. Thus, $\rightsquigarrow_O$ does not generate anything that corresponds to $be \leftrightarrow de$. □

This example can be generalized to provide a simple analysis of simplification steps, one that shows which steps correspond to $\rightsquigarrow_O$-inferences and which do not. Assume we have two rewrite rules $AB \to A$ and $CD \to C$, corresponding to non-trivial Horn clauses ($|B| = 1$, $B \not\subseteq A$, $|D| = 1$, $D \not\subseteq C$), and such that $CD \to C$ simplifies $AB \to A$. We distinguish three cases:

1. In the first one, CD appears in AB because CD appears in A. In other words, $A = CDE$ for some E. Then, the simplification step is

   $$\frac{CDEB \rightarrow CDE,\ CD \rightarrow C}{\overline{CEB \rightarrow CE,\ CD \rightarrow C}}$$

   (where simplification is actually applied to both sides). The corresponding implications are $A \Rightarrow B$ and $C \Rightarrow D$. Since $A \Rightarrow B$ is $CDE \Rightarrow B$, implicational overlap applies to generate the implication $CE \Rightarrow B$ that corresponds to $CEB \rightarrow CE$:

   $$\frac{C \Rightarrow D,\ CDE \Rightarrow B}{CE \Rightarrow B}\ .$$

   The isotony rule applied to $CE \Rightarrow B$ and $CDE \Rightarrow B$ reduces the latter to $CDE \Rightarrow \emptyset$, which is deleted by definiteness: a combination of implicational overlap, isotony and definiteness simulates the effects of simplification.
2. In the second case, CD appears in AB because C appears in A, that is, $A = CE$ for some E, and $D = B$. Then, the simplification step is

   $$\frac{CEB \rightarrow CE,\ CB \rightarrow C}{\overline{CE \leftrightarrow CE,\ CB \rightarrow C}}\ ,$$

   and $CE \leftrightarrow CE$ is removed by deletion. The isotony inference

   $$\frac{C \Rightarrow B,\ CE \Rightarrow B}{\overline{C \Rightarrow B,\ CE \Rightarrow \emptyset}}\ ,$$

   generates $CE \Rightarrow \emptyset$ which gets deleted by definiteness.
3. The third case is the generalization of Example 8: CD appears in AB because D appears in A, and C is made of B and some F that also appears in A, that is, $A = DEF$ for some E and F, and $C = BF$. The simplification step is

   $$\frac{DEFB \rightarrow DEF,\ BFD \rightarrow BF}{\overline{BFE \leftrightarrow DEF,\ BFD \rightarrow BF}}\ .$$

   Implicational overlap applies

   $$\frac{DEF \Rightarrow B,\ BF \Rightarrow D}{DEF \Rightarrow D}$$

   to generate an implication that is first reduced by extensiveness to $DEF \Rightarrow \emptyset$ and then eliminated by definiteness. Thus, nothing corresponding to $BFE \leftrightarrow DEF$ gets generated.

It follows that whatever is generated by $\leadsto_O$ is generated by $\leadsto_R$, but may become redundant eventually:

Theorem 3. *For every implicational system S, for all fair and contracting derivations $S = S_0 \leadsto_O S_1 \leadsto_O \cdots$ and $R_S = R_0 \leadsto_R R_1 \leadsto_R \cdots$, for all $FG \to F \in R_{(S_\infty)}$, either $FG \to F \in (R_S)_\infty$ or $FG \to F$ is redundant in $(R_S)_\infty$.*

Proof. For all $FG \to F \in R_{(S_\infty)}$, $F \Rightarrow G \in S_\infty$ by Definition 3, and $F \Rightarrow G \in S_j$ for some $j \geq 0$. Let j be the smallest such index. If $j = 0$, or $S_j = S$, $FG \to F \in R_S = R_0$ by Definition 3. If $j > 0$, $F \Rightarrow G$ was generated by an application of implicational overlap, the isotony rule or extensiveness. By Lemma 3 and the assumption that the $\leadsto_R$-derivation is fair and contracting, $FG \to F \in R_k$ for some $k > 0$. In both cases, $FG \to F \in R_k$ for some $k \geq 0$. If $FG \to F$ persists, then $FG \to F \in (R_S)_\infty$. Otherwise, $FG \to F$ gets rewritten by simplification and is, therefore, redundant in $(R_S)_\infty$. □

Since the limit of a fair and contracting $\leadsto_O$-derivation is $O(S)$, it follows that:

Corollary 3. *For every implicational system S, for all fair and contracting derivations $S = S_0 \leadsto_O S_1 \leadsto_O \cdots$ and $R_S = R_0 \leadsto_R R_1 \leadsto_R \cdots$, and for all $FG \to F \in R_{O(S)}$, either $FG \to F$ is in $(R_S)_\infty$ or else $FG \to F$ is redundant in $(R_S)_\infty$.*

7 Discussion

Although simple from the point of view of computational complexity,[1] propositional Horn theories, or, equivalently, Moore families presented by implicational systems, appear in many fields of computer science. In this article, we analyzed the notions of direct and direct-optimal implicational system in terms of completion and canonicity. We found that a direct implicational system corresponds to the canonical limit of a derivation by completion that features expansion by equational overlap and contraction by forward simplification. When completion also features backward simplification, it computes the image of a given set with respect to the closure operator associated with the given implicational system. In other words, it computes the minimal model that satisfies both the implicational system and the set. On the other hand, a direct-optimal implicational system does not correspond to the limit of a derivation by completion, because the underlying proof orderings are different and, therefore, normalization induces two different notions of optimization. Accordingly, we introduced a new notion of optimality for implicational systems, termed *rewrite optimality*, that corresponds to canonicity defined by completion up to redundancy.

Directions for future work include generalizing this analysis beyond propositional Horn theories, studying enumerations of Moore families and related structures (see [10] and Sequences A102894–7 and A108798–801 in [18]), and exploring connections between canonical systems and decision procedures, or the rôle of canonicity of presentations in specific contexts where Moore families occur, such as in the abstract interpretations of programs.

[1] Satisfiability, and hence entailment, can be decided in linear time [13].

References

1. Bachmair, L., Dershowitz, N.: Equational inference, canonical proofs, and proof orderings. Journal of the ACM 41(2), 236–276 (1994)
2. Bertet, K., Monjardet, B.: The multiple facets of the canonical direct implicational basis. Cahiers de la Maison des Sciences Economiques b05052, Université Paris Panthéon-Sorbonne (June 2005), http://ideas.repec.org/p/mse/wpsorb/b05052.html
3. Bertet, K., Nebut, M.: Efficient algorithms on the Moore family associated to an implicational system. Discrete Mathematics and Theoretical Computer Science 6, 315–338 (2004)
4. Bonacina, M.P., Dershowitz, N.: Abstract canonical inference. ACM Transactions on Computational Logic 8(1), 180–208 (2007)
5. Bonacina, M.P., Hsiang, J.: On rewrite programs: Semantics and relationship with Prolog. Journal of Logic Programming 14(1 & 2), 155–180 (1992)
6. Bonacina, M.P., Hsiang, J.: Towards a foundation of completion procedures as semidecision procedures. Theoretical Computer Science 146, 199–242 (1995)
7. Caspard, N., Monjardet, B.: The lattice of Moore families and closure operators on a finite set: A survey. Electronic Notes in Discrete Mathematics 2 (1999)
8. Darwiche, A.: Searching while keeping a trace: the evolution from satisfiability to knowledge compilation. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130. Springer, Heidelberg (2006)
9. Dershowitz, N.: Computing with rewrite systems. Information and Control 64(2/3), 122–157 (1985)
10. Dershowitz, N., Huang, G.-S., Harris, M.A.: Enumeration problems related to ground Horn theories, http://arxiv.org/pdf/cs.LO/0610054
11. Dershowitz, N., Jouannaud, J.-P.: Rewrite systems. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, vol. B, pp. 243–320. Elsevier, Amsterdam (1990)
12. Dershowitz, N., Marcus, L., Tarlecki, A.: Existence, uniqueness, and construction of rewrite systems. SIAM Journal of Computing 17(4), 629–639 (1988)
13. Dowling, W.F., Gallier, J.H.: Linear-time algorithms for testing the satisfiability of propositional Horn formulæ. Journal of Logic Programming 1(3), 267–284 (1984)
14. Furbach, U., Obermaier, C.: Knowledge compilation for description logics. In: Dershowitz, N., Voronkov, A. (eds.) LPAR 2007. LNCS (LNAI), vol. 4790, Springer, Heidelberg (2007)
15. Horn, A.: On sentences which are true of direct unions of algebras. Journal of Symbolic Logic 16, 14–21 (1951)
16. McKinsey, J.C.C.: The decision problem for some classes of sentences without quantifiers. Journal of Symbolic Logic 8, 61–76 (1943)
17. Roussel, O., Mathieu, P.: Exact knowledge compilation in predicate calculus: the partial achievement case. In: McCune, W. (ed.) CADE 1997. LNCS, vol. 1249, pp. 161–175. Springer, Heidelberg (1997)
18. Sloane, N.J.A.: The On-Line Encyclopedia of Integer Sequences (1996-2006), http://www.research.att.com/~njas/sequences

Challenges in the Automated Verification of Security Protocols

Hubert Comon-Lundh

Ecole Normale Supérieure de Cachan and
Research Center for Information Security,
National Institute of Advanced
Industrial Science and Technology (AIST)
Tokyo, Japan
h.comon-lundh@aist.go.jp

Abstract. The application area of security protocols raises several problems that are relevant to automated deduction. We describe in this note some of these challenges.

1 Introduction

Security protocols are small distributed programs, aiming at securely achieving some transaction, while relying on a public or a non-reliable network. Getting formal guarantees that such protocols satisfy their specifications is an important issue, because of the numerous applications and the economical and societal impact of potential failures.

Program testing is irrelevant in this context, because of the malicious nature of attacks. On the opposite, security protocols constitute an ideal field of applications for research in automated deduction: the protocols are small, as well as their specification, yet their verification is non-trivial. The purpose of this note is to show several interesting issues in automated deduction, which are raised by this verification problem.

We will focus on problems that can (or could) be formulated in first-order logic or equational logic. Already in early works, first-order clausal formalisms have been used for security protocols [45] and intruder capabilities formalized using term rewriting systems [47]. Since then, there has been a large amount of work in the area, which *we do not survey here* (in particular many relevant references are certainly missing). Despite this research, there are still open questions, some of which are described in this note.

We will organize the note stepwise, starting with the most elementary notions (intruder deduction) in section 2 and enrich the model in further sections, each time reporting some unsolved problem. In section 3, we consider the security problem for a fixed number of protocol sessions. In section 4, we consider resolution-based theorem proving applied to security protocols. In section 5, we consider security properties that are usually defined as indistinguishability between two processes. Finally, we consider modularity issues in section 6.

2 Intruder Deductions

The security of a protocol depends on several inputs: the protocol, the security property, but also the intruder capabilities and the (assumed) algebraic properties of the security primitives. We assume that $\mathcal{F}$ is a set of function symbols representing all security primitives and data. $\mathcal{A}$ is a set of equations, typically associativity and commutativity of some symbols in $\mathcal{F}$ and $\mathcal{R}$ is a rewrite system, convergent modulo $\mathcal{A}$ capturing the expected computations on messages. A typical rule in $\mathcal{R}$ would be $\mathsf{dec}(\mathsf{enc}(x, y), y^{-1}) \to x$.

In the simplest setting, we only consider offline attacks: we forget how an attacker got messages and simply assume (s)he holds a set of messages. In addition, we only consider security properties that can be modeled as the confidentiality of some piece of a message.

We also assume here that the security primitives, such as encryption algorithms, do not leak any information besides what is modeled through the algebraic properties: we consider a formal model, that is sound with respect to the computational one.

In such a (restricted) case, security is an *Entscheidungsproblem* for a particular deduction system, representing the attacker computing capabilities. We let $\mathcal{F}_{\text{pub}}$ be the subset of $\mathcal{F}$ consisting of *public symbols*. Messages are represented as terms in the algebra $T(\mathcal{F})$ (or possibly a quotient $T(\mathcal{F})/_{=_\mathcal{A}}$) and the inference rules are of the form

$$\frac{s_1 \ \ldots \ s_n}{f(s_1, \ldots, s_n)\downarrow} R_f$$

The attacker may apply the function $f \in \mathcal{F}_{\text{pub}}$ to messages $s_1, \ldots, s_n$, getting $f(s_1, \ldots, s_n)\downarrow$, the normal form of $f(s_1, \ldots, s_n)$ with respect to $\mathcal{R}$.

Example 1. The most typical example is the *perfect cryptography*, for symmetric encryption. $\mathcal{F}_{\text{pub}}$ contains pairing: $< s_1, s_2 >$ is the pair of the two terms s_1, s_2, encryption: $\{s\}_k$ is the encryption of s with k, decryption dec, projections π_1, π_2 and some public constants. $\mathcal{F} = \mathcal{F}_{\text{pub}} \cup \mathcal{S}$ where $\mathcal{S}$ is a set of (secret) constants, disjoint from $\mathcal{F}_{\text{pub}}$. $\mathcal{A}$ is empty and the rewrite system consists of three rules:

$$\mathsf{dec}(\{s\}_k, k) \to s \qquad \pi_1(< x, y >) \to x \qquad \pi_2(< x, y >) \to y$$

Many examples of (other) relevant equational theories and rewrite systems are given in [33].

The set of inference rules needs not be finite, as, for variadic symbols (or associative-commutative symbols), we may want one version of the rule for each value of n. It is also convenient sometimes to consider combinations of function symbols instead of a single function application.

Also note the resulting term $f(s_1, \ldots, s_n)\downarrow$ is assumed to be a message: the set of messages M is a subset of $T(\mathcal{F})$ (or $T(\mathcal{F})/_{=_\mathcal{A}}$) and a valid instance of the rule assumes that both the premisses and the conclusion are in M. For instance, in example 1, terms in M do not contain the symbols $\mathsf{dec}, \pi_1, \pi_2$ and, if the decryption does not succeed, the attacker does not learn anything about the plaintext.

The set of inference rules yields a deducibility relation $\vdash \subseteq 2^M \times M$, using repeatedly the inference rules from an initial knowledge $H \subseteq 2^M$.

In most relevant cases, as shown in [29], an equivalent inference system can be obtained by "inlining" the rewriting rules applications, getting rid of the normalization symbol; the rewriting steps in the conclusion are replaced with finitely many patterns in the premisse. In example 1, we would get the following inference rules:

$$\frac{x_1 \quad x_2}{\{x_1\}_{x_2}} \qquad \frac{x_1 \quad x_2}{< x_1, x_2 >} \qquad \frac{\{x_1\}_{x_2} \quad x_2}{x_1} \qquad \frac{< x_1, x_2 >}{x_1} \qquad \frac{< x_1, x_2 >}{x_2}$$

Definition 1. *Given an intruder deduction system, the* deducibility problem *is, given a finite subset H of M and a term $s \in M$, to decide whether $H \vdash s$.*

This may also be formulated using rewriting systems: if we let $H = \{t_1, \ldots, t_n\}$ and σ be the substitution $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$, the problem is equivalent to:

Given σ, is there a term $\zeta \in T(\mathcal{F}_{\text{pub}}, \{x_1, \ldots, x_n\})$ such that $\zeta\sigma \downarrow =_{\mathcal{A}} s$.

Now, what is a bit surprising at first sight is that many of such intruder deduction systems have very nice properties: the deducibility problem is in PTIME. This relies on a subformula property of the deduction system: if the function symbols applications are gathered together in an appropriate way, then for any deducible s, there is a proof in which intermediate steps belong to $\mathsf{St}(s, H)$, the subterms of the hypotheses and the conclusion. In such a case, we say that the deduction system is *local*, following [44]. Then, if one step deduction is itself decidable in PTIME, the deducibility problem can be reduced to a simple fixed point computation of the deducible subterms.

Such a locality theorem has been established for many examples of intruder deduction systems, for instance perfect cryptography, for symmetric and asymmetric encryption and signatures, commutative encryption, ECB and CBC encryption modes, blind signatures, exclusive or [22], Abelian groups [21], modular exponentiation, and others [35,19]. Actually, each time a new protocol is designed, which relies on a new equational theory, a new locality result is established.

Instead of repeating the work for each new theory, we wish to get a more general result. This raises the following challenge:

Challenge 1. *For which classes of (AC)-rewrite systems, do we get a locality property? A decidable deducibility problem?*

Related Works

There are actually some works [30,5], answering, at least partly, this question, *when $\mathcal{A}$ is empty.* Many examples however involve associative and commutative symbols. So, the above challenge can be seen as extending these results to the AC case.

[8] also provides with a general criterion, amenable to automated locality proofs. First, locality is generalized, replacing the subterm relation with any well-founded ordering: $\mathsf{St}(s, H)$ above is replaced with the set of terms that are smaller than s, H. Of course, if we want to derive a decision procedure, this set must be finite. Next, they show that non-local inference systems can be completed into local ones, using ordered resolution (*saturated sets* are local). However, this process may not terminate. That is what happens when, in our applications, $\mathcal{A}$ is not empty. Typically, we need to deal with infinitely many clauses of the form $I(x_1), \ldots, I(x_n) \rightarrow I(x_1 + \ldots + x_n)$ when $+$ is associative and commutative. Answering the challenge along this line of research would require to extend the results of [8] to some class of infinite sets of clauses.

Other Interesting Issues

Another related problem is the *combination* of intruder theories: If the deducibility problem is decidable for two intruder theories, under which condition is it decidable in the union of the two theories ? Similar questions may be asked for local theories.

This is an important issue when the deduction system is large. For instance in [19], the authors consider an intruder deduction system, which includes several associative-commutative symbols and a lot of rules. This system formalizes some arithmetic properties, which must be considered in the model, since they are used by the participants of an actual electronic purse protocol. The locality proof of [19] is a direct (tedious) proof and would benefit from combination results.

There is a combination result in the disjoint case [7], however not applicable to the above example.

3 Bounded Number of Sessions

Now, we enrich the model, considering some active attacks: the attacker may not only perform offline deductions, but also send fake messages and take advantage of the reply. We assume however that he can send only a fixed number of such messages (this number is given as a parameter).

One of the main problems, which yields "man-in-the-middle attacks" such as the famous attack of [42], is that pieces of messages cannot be analysed by some honest parties, hence could be replaced by arbitrary messages.

Example 2. An honest agent a generates a random number n and sends it, together with his identity, encrypted with the public key of another honest agent b. Such a message can be represented as $\{< a, n >\}_{\mathsf{pub}(b)}$, a (ground) term in our algebra $T(\mathcal{F})$. However, b's view is different. Upon receiving this message, (s)he decrypts and finds a pair, whose second component is an arbitrary message. Suppose that b has to send back $\{< b, n >\}_{\mathsf{pub}(a)}$, the corresponding rule for b will be:

Upon receiving any $\{< x, y >\}_{\mathsf{pub}(b)}$ reply by sending $\{< b, y >\}_{\mathsf{pub}(x)}$, maybe after checking that x is the identity of somebody with whom b is willing to communicate.

The important issue here is that y can be replaced by any message and the attacker may take advantage of this.

Hence, a simple class of protocols can be defined by finitely many *roles*, which are again finite sequences of rules $s \Rightarrow t$, meaning that, upon reception of a message $s\sigma$, the agent replies $t\sigma$. In general, each role also generates new names (keys or nonces) and may have a more complex control structure, for instance specifying what to do when the message does not match the expected s. In this section, we will not consider replication of the roles, hence new names can simply be distinguished using distinct constants and we only consider a simple linear structure of the control. Even more, by guessing an ordering between the different actions of each role, we get a sequence $s_i \Rightarrow t_i$ of query $\Rightarrow$ response (such as $\{< x, y >\}_{\mathsf{pub}(b)} \Rightarrow \{< b, y >\}_{\mathsf{pub}(x)}$ in example 2),

In this setting, there is an attack on the secrecy of s, if there is a substitution σ such that

$$\begin{aligned} S_0 &\vdash s_1\sigma \\ S_0, t_1\sigma &\vdash s_2\sigma \\ &\vdots \\ S_0, t_1\sigma, \ldots, t_{n-1}\sigma &\vdash s_n\sigma \\ S_0, t_1\sigma, \ldots, t_n\sigma &\vdash s \end{aligned}$$

where S_0 is the set of initial intruder knowledge and s is the data supposed to remain secret. $s_1\sigma, \ldots, s_n\sigma$ are the successive (fake) messages constructed by the attacker and $t_1\sigma, \ldots, t_n\sigma$ are the successive messages sent by the agents. The left members of the deducibility constraints increase as the attacker's gets new messages from the network. Without the last constraint, there is a trivial substitution σ_0, corresponding to the normal execution of the protocol, in which the attacker simply forwards the messages without modifying them : $s_{i+1}\sigma_0 = t_i\sigma_0$ for every $i < n$.

Equivalently, we can define an attack using term rewriting systems: there is an attack if there are terms $\zeta_0, \ldots, \zeta_n \in T(\mathcal{F}_{\text{pub}} \cup S_0, \{x_1, \ldots, x_n\})$ and a substitution σ such that

- For every $0 \leq i \leq n-1$, $(\zeta_i\{x_1 \mapsto t_1, \ldots, x_i \mapsto t_i\})\sigma\downarrow =_{\mathcal{A}} s_{i+1}\sigma\downarrow$
- $(\zeta_n\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\})\sigma\downarrow =_{\mathcal{A}} s$

$\zeta_0, \ldots, \zeta_n$ are the *recipes*, which constitute the attacker's strategy: they specify how to combine the known messages to produce the next fake message.

Deciding if there is an attack consists then in deciding whether such a substitution σ exists. This problem has been shown to be in NP for a number of intruder theories [49,23,22,21,46].

There are also a number of results for classes of theories, for instance: subterm theories [11] and monoidal theories [37]. This does not cover however some relevant examples such as the one described in [19].

In most (if not all) these results, the key is a *small attack property* (also called *conservativity* in e.g. [37]): if there is an attack then there is an attack, which is built, stacking pieces of protocol rules. Then, the search space for σ becomes finite, yielding decision algorithms.

This property is not well-understood; the existence of small attacks can be recasted as the existence of particular (small) proofs: there should be proof simplification rules, which justify such a property. This yields the following challenge:

Challenge 2. *Give a proof-theoretic explanation of the small attack property.*

Related Work

A partial answer is given in [14]: the authors give proof simplification rules, which are strongly normalizing. The normal proofs only correspond to small attacks. However, this work only covers a restricted class of intruder deduction rules, assuming $\mathcal{A}$ is empty, as well as additional syntactic restrictions.

An approach along the lines of [8] does not work here: the clausal formalisation of the protocol rules introduces some over-approximations, as explained in the next section.

Other Interesting Issues

Again, combination of decision procedures for a bounded number of sessions is an important issue, as, in general, many protocols are running at the same time on the same device. This problem is addressed in [24,25], but there are still examples of relevant complex equational theories that cannot (and should) be broken into pieces using these results (see e.g. [19]). Hence, there is still some work to be done in this area. (See also section 6 for the general modularity problem, when the number of sessions is unbounded.)

4 Clausal Theorem Proving and Security Protocols

In the automated deduction community, it has become a habit to recast all decision results as terminating resolution strategies [39]. This allows in principle to use a single clausal theorem prover for any of the particular decision problems. What about the decision results of the last section? This will be our next challenge, but let us first introduce and clarify the problems.

There are several formalizations of security protocols within first-order logic. After C. Meadows [45], C. Weidenbach in [50] was one of the first who observed that resolution methods are quite successful in this area. Then the security analyzer ProVerif [15,16] is also a resolution-based prover, and one of the most successful protocol verifiers.

Such formalizations rely on approximations. Otherwise, an accurate model such as [28] is hopeless, as far as termination is concerned. Among the approximations:

– Nonces, i.e., new names that are generated at each session of the protocol, are represented as terms, depending on the environment in which they are generated. This over-approximation is always correct as long as secrecy properties are considered. As already mentioned, for a bounded number of sessions, we can simply represent these nonces using different names.
– The ordering in which messages of a given role are played is lost in the clausal translation. This can be recovered in case of a bounded number of sessions, recording and verifying the ordering of events.
– Clauses are universally quantified formulas. This is fine for clauses representing the attacker's capabilities, such as

$$I(x), I(y) \rightarrow I(< x, y >)$$

as the attacker may re-use such a construction any number of times. This might be more problematic for protocol rules as, when we use variables in place of messages (for un-analysable parts), the attacker may choose any value, but has to commit to this value: the rule should not be replayed.

This last problem is well-known in automated deduction (typically in tableau methods): we need *rigid variables*. While rigid variables are still universally quantified, they can get only one instance: each rigid clause can be instanciated only once and then used as many times as we wish. Using a mixture of rigid clauses and flexible clauses is therefore more accurate for the security protocols verification. This has been investigated in [38].

We may however simply translate the rigid clauses into first-order clauses, at the price of adding new arguments, while preserving satisfiability [4]. We get in this case a faithful translation into first-order clauses of the security problem for a bounded number of sessions.

Challenge 3. *Design a resolution strategy, which yields a NP-decision method for the formulas corresponding to bounded sessions protocol security.*

Related Work

Coming back to the over-approximated problem for an unbounded number of sessions, resolution methods are unusually successful: in many cases, the deduction process terminates. There are several works explaining this phenomenon. For instance, restrictions on the variables occurring in each protocol clause, the protocol model falls into a decidable fragment of first-order logic [34,27,26]. Also, assuming some tagging of (sub)messages yiedls termination [18,48].

Other Interesting Issues

They include speeding-up the deduction, relying on constraints [9] or strategies [16] or including general (user-defined) equational theories [10,29,17].

5 Proofs of Equivalences

Many security properties cannot be directly modeled in first-order logic, since they are not properties of a single execution trace. They are better stated as indistinguishability properties between two processes. Typical such examples are anonymity and (strong) secrecy; while secrecy (or confidentiality) requires a given message s to remain unknown to the attacker, strong secrecy requires that an attacker cannot learn anything about s. The latter is expressed by an indistinguishability property between the real protocol and the protocol in which s is replaced by a new name. Properties of electronic voting protocols are also typically expressed in this way [36].

In case of equivalence properties, the deduction problem (definition 1) is replaced by deciding *static equivalence*:

Definition 2. *Two sequences of ground terms* $s_1, \ldots, s_n$ *and* $t_1, \ldots, t_n$ *are* statically equivalent*, which we write* $s_1, \ldots, s_n \sim t_1, \ldots, t_n$*, if, for any terms* $u, v \in T(\mathcal{F}, \{x_1, \ldots, x_n\})$*,*

$$\begin{gathered} u\{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\}\downarrow =_{\mathcal{A}} v\{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\}\downarrow \\ \Updownarrow \\ u\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}\downarrow =_{\mathcal{A}} v\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}\downarrow \end{gathered}$$

The decision of static equivalence has been shown for several term rewriting systems [1,32] and has been implemented for a wide class of equational theories [13,17].

Consider now protocols. Each role can be described by a simple sequential process, for instance in the applied π-calculus [2] and each protocol is expressed as a parallel composition of a fixed number of roles $P_1, \ldots, P_n$ (resp. $Q_1, \ldots, Q_n$). The security properties that we wish to consider are expressed as the observational equivalence

$$!(P_1 \| \cdots \| P_n) \approx !(Q_1 \| \cdots \| Q_n)$$

In words: an attacker cannot distinguish between the two versions of the protocol. The exclamation mark is a replication: we may consider any number of copies of each protocol roles.

In the bounded sessions version of the problem, we simply remove the exclamation marks, and we can replace all fresh names by distinct constants. We call *bounded protocols* the resulting processes. As before, any interleaving τ_P of actions of the process $P = P_1 \| \cdots \| P_n$ corresponds to a sequence of query $\Rightarrow$ response: $s_i \Rightarrow t_i$, for $i = 1, ..., m$. For any such sequence, a *valid attacker on* τ_P consists in a substitution σ and terms $\zeta_0, \ldots, \zeta_{m-1} \in T(\mathcal{F}_{\text{pub}} \cup S_0, \{x_1, \ldots, x_{m-1}\})$ such that, for every i, $\zeta_i \theta_i \sigma \downarrow =_{\mathcal{A}} s_{i+1} \sigma \downarrow$, where θ_i is the substitution $\{x_1 \mapsto t_1, \ldots, x_i \mapsto t_i\}$ (the empty substitution when $i = 0$). This matches our definitions in section 3: $\zeta_0, \ldots, \zeta_{m-1}$ are the recipes specifying the attacker's strategy.

Now, the bounded protocols P and Q are *equivalent* if, for any interleaving τ_P of actions of P, for any valid attacker on τ_P $(\zeta_0, \ldots, \zeta_{m-1}, \sigma)$, there is an interleaving τ_Q of actions of Q and a substitution σ' such that $(\zeta_0, \ldots, \zeta_{m-1}, \sigma')$ is a valid attacker on τ_Q and $t_1\sigma, \ldots, t_m\sigma \sim t'_1\sigma', \ldots, t'_m\sigma'$ where $t_1, \ldots, t_m$ is the sequence of responses in τ_P and $t'_1, \ldots, t'_m$ is the sequence of responses in τ_Q (and conversely, exchanging P and Q). In words: P and Q are equivalent if, whatever the attacker does, the sequence of replies from P is statically equivalent to the corresponding sequence of replies of Q.

All decision results reported in section 3 only concern the secrecy property. We would like to extend them to more security properties:

Challenge 4. *In case of a bounded number of sessions, give decision and complexity results for the equivalence of protocols.*

The problem is more complex than in the secrecy case, as we cannot guess first a trace and then solve a finite deducibility constraint system.

Related Work

As far as we know, the only related works are

– a decision result for the spi-calculus (and no equational theory) [40] and
– a decision result for an approximate equivalence (implying observational equivalence) and subterm-convergent theories [12].

Other Interesting Issues

If we consider an arbitrary number of sessions, in [17], the authors design a coding of pairs of processes into *bi-processes*. In ProVerif, this coding itself is translated into first order-clauses, yielding automatic proofs of equivalences, for an unbounded number of sessions. To the best of our knowledge, this is the only tool, which allows for automatic equivalence proofs for an unbounded number of sessions. The encoding into bi-processes requires however an approximation of bisimulation relations: two processes might be equivalent and yet there is no equivalence proof in ProVerif. An interesting open question is to find a class of protocols/properties for which the bi-process technique is complete.

6 Modularity

The problem of combining protocols is an important issue. Usually, several protocols may run on the same device and it is important that, once each single protocol is proved secure, they remain secure when they run concurrently.

There are more reasons for studying this problem of security proofs modularity, which are coming from another area of computer science. In the area of computational security, usually related to cryptology, a huge amount of work has been devoted to *universal composability* [20,41]. The idea is to get modularity theorems (called *composition theorems* in this context), which show that the security for one session implies the security for an unbounded number of sessions and that two secure protocols can run concurrently, without generating new attacks. However, both the composition theorems and the proofs that security primitives satisfy the hypotheses of these theorems are quite delicate in the computational world [41]. Even more, to the best of our knowledge, there is no composition result in the computational setting, which would allow some secrets (such as a pair <public key, private key>) to be shared by protocols. Using a single public key in several protocols is however a common practice (and actually desirable).

In order to get computational security guarantees, there is an alternative method: using *soundness theorems* in the style of [3], the computational models are faithfully abstracted into symbolic ones. Then, proving the security at the computational level is reduced to proving a corresponding appropriate property at the symbolic level, which might be automated. There is a large body of literature on soundness theorems, which we do not review here.

Assuming appropriate soundness results, the modularity of symbolic security proofs would have therefore an important impact in the computational security community.

To be more precise, a modularity result in the symbolic setting would state that for a family of shared names $\overline{n}$ (typically the public and private keys), a class of protocols, a class of intruder deduction systems and a class of security properties, the security of two individual protocols P_1 and P_2 implies the security of the combined protocol $P_1 \| P_2$.

Challenge 5. *For which intruder deduction systems and security properties can we get modularity theorems?*

Additionally, for applications to universal composability, but also in order to get decision results, we wish to get some kind of modularity of a protocol with itself: assuming that secrets such as pairs of public and private keys are shared over sessions, the security of P should imply the security of $!P$.

Challenge 6. *For which classes of protocols, intruder theories and security properties does the security for a single session imply the security for an arbitrary number of sessions?*

Related Work

A recent result [31] proves a modularity theorem for a wide class of protocols. However, it only considers the secrecy property and the intruder deduction system is fixed to the simple rules of example 1 for symmetric and asymmetric encryption. Generalizing the result to equivalence properties and other intruder systems is an open question.

The proof of this result relies on several techniques from automated deduction:

- The small attack property (see section 3),
- Properties of unification and deducibility constraints,
- An analog of Craig's interpolation theorem for intruder deduction systems.

Concerning the challenge 6, a first result was proved by G. Lowe [43], however for a restricted class of protocols. For instance [43] assumes that any two distinct terms headed with encryption, that occur in the protocol, are not unifiable. A similar result is shown in [6] for another class of protocols. However, in both cases, only secrecy is considered and the simple intruder deduction system is the one of example 1 (for symmetric and asymmetric encryption).

Acknowledgments

Many thanks to Reynald Affeldt, David Basin, Bruno Blanchet, Véronique Cortier, Stéphanie Delaune, Michael Rusinowitch for their comments on an early version of this note.

References

1. Abadi, M., Cortier, V.: Deciding knowledge in security protocols under equational theories. Theoretical Computer Science 367(1–2), 2–32 (2006)
2. Abadi, M., Fournet, C.: Mobile values, new names, and secure communication. In: Proceedings of the 28th ACM Symposium on Principles of Programming Languages (POPL 2001), January 2001, pp. 104–115 (2001)
3. 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. Springer, Heidelberg (2000)
4. Affeldt, R., Comon-Lundh, H.: First-order logic and security protocols (unpublished manuscript) (April 2008)
5. Anantharaman, S., Narendran, P., Rusinowitch, M.: Intruders with caps. In: Baader, F. (ed.) RTA 2007. LNCS, vol. 4533. Springer, Heidelberg (2007)
6. Arapinis, M., Delaune, S., Kremer, S.: From one session to many: Dynamic tags for security protocols. Research Report LSV-08-16, Laboratoire Spécification et Vérification, ENS Cachan, France (May 2008)
7. Arnaud, M., Cortier, V., Delaune, S.: Combining algorithms for deciding knowledge in security protocols. In: Konev, B., Wolter, F. (eds.) FroCos 2007. LNCS (LNAI), vol. 4720. Springer, Heidelberg (2007)
8. Basin, D., Ganzinger, H.: Automated complexity analysis based on ordered resolution. Journal of the Association of Computing Machinery 48(1), 70–109 (2001)
9. Basin, D., Mödersheim, S., Viganò, L.: Constraint Differentiation: A New Reduction Technique for Constraint-Based Analysis of Security Protocols. In: Proceedings of CCS 2003, pp. 335–344. ACM Press, New York (2003)
10. Basin, D., Mödersheim, S., Viganò, L.: Algebraic intruder deductions. In: Sutcliffe, G., Voronkov, A. (eds.) LPAR 2005. LNCS (LNAI), vol. 3835. Springer, Heidelberg (2005)
11. Baudet, M.: Deciding security of protocols against off-line guessing attacks. In: Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS 2005), Alexandria, Virginia, USA, November 2005, pp. 16–25. ACM Press, New York (2005)

12. Baudet, M.: Sécurité des protocoles cryptographiques : aspects logiques et calculatoires. Thèse de doctorat, Laboratoire Spécification et Vérification, ENS Cachan, France (January 2007)
13. Baudet, M.: Yapa: Yet another protocol analyser (2008), http://www.lsv.ens-cachan.fr/~baudet/yapa/index.html
14. Bernat, V., Comon-Lundh, H.: Normal proofs in intruder theories. In: Revised Selected Papers of the 11th Asian Computing Science Conference (ASIAN 2006). LNCS, vol. 4435. Springer, Heidelberg (2008)
15. Blanchet, B.: An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In: 14th IEEE Computer Security Foundations Workshop (CSFW-14), Cape Breton, Nova Scotia, Canada, June 2001, pp. 82–96. IEEE Computer Society, Los Alamitos (2001)
16. Blanchet, B.: An automatic security protocol verifier based on resolution theorem proving (invited tutorial). In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632. Springer, Heidelberg (2005)
17. Blanchet, B., Abadi, M., Fournet, C.: Automated verification of selected equivalences for security protocols. Journal of Logic and Algebraic Programming 75(1), 3–51 (2008)
18. Blanchet, B., Podelski, A.: Verification of cryptographic protocols: Tagging enforces termination. Theoretical Computer Science 333(1–2), 67–90 (2005)
19. Bursuc, S., Comon-Lundh, H., Delaune, S.: Deducibility constraints, equational theory and electronic money. In: Comon-Lundh, H., Kirchner, C., Kirchner, H. (eds.) Jouannaud Festschrift. LNCS, vol. 4600. Springer, Heidelberg (2007)
20. Canetti, R., Rabin, T.: Universal composition with joint state. Cryptology ePrint Archive, report 2002/47 (November 2003)
21. Chevalier, Y., Küsters, R., Rusinowitch, M., Turuani, M.: Deciding the security of protocols with Diffie-Hellman exponentiation and products in exponents. In: Pandya, P.K., Radhakrishnan, J. (eds.) FSTTCS 2003. LNCS, vol. 2914. Springer, Heidelberg (2003)
22. Chevalier, Y., Küsters, R., Rusinowitch, M., Turuani, M.: An NP decision procedure for protocol insecurity with XOR. In: Eighteenth Annual IEEE Symposium on Logic in Computer Science (LICS) (2003)
23. Chevalier, Y., Küsters, R., Rusinowitch, M., Turuani, M.: Deciding the security of protocols with commuting public key encryption. In: Proc. Workshop on Automated Reasoning for Security Protocol Analysis (ARSPA). Electronic Notes in Theoretical Computer Science, vol. 125 (2004)
24. Chevalier, Y., Rusinowitch, M.: Combining Intruder Theories. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580. Springer, Heidelberg (2005)
25. Chevalier, Y., Rusinowitch, M.: Hierarchical Combination of Intruder Theories. In: Pfenning, F. (ed.) RTA 2006. LNCS, vol. 4098, pp. 108–122. Springer, Heidelberg (2006)
26. H. Comon and V. Cortier. Tree automata with one memory, set constraints and cryptographic protocols. Theoretical Computer Science, 331(1):143–214, Feb. 2005.
27. Comon-Lundh, H., Cortier, V.: New decidability results for fragments of first-order logic and application to cryptographic protocols. In: Nieuwenhuis, R. (ed.) RTA 2003. LNCS, vol. 2706. Springer, Heidelberg (2003)
28. Comon-Lundh, H., Cortier, V.: Security properties: Two agents are sufficient. Science of Computer Programming 50(1-3), 51–71 (2004)

29. Comon-Lundh, H., Delaune, S.: The finite variant property: How to get rid of some algebraic properties. In: Giesl, J. (ed.) RTA 2005. LNCS, vol. 3467. Springer, Heidelberg (2005)
30. Comon-Lundh, H., Treinen, R.: Easy intruder deductions. In: Dershowitz, N. (ed.) Verification: Theory and Practice. LNCS, vol. 2772. Springer, Heidelberg (2004)
31. Cortier, V., Delaitre, J., Delaune, S.: Safely composing security protocols. In: Arvind, V., Prasad, S. (eds.) FSTTCS 2007. LNCS, vol. 4855. Springer, Heidelberg (2007)
32. Cortier, V., Delaune, S.: Deciding knowledge in security protocols for monoidal equational theories. In: Dershowitz, N., Voronkov, A. (eds.) LPAR 2007. LNCS (LNAI), vol. 4790. Springer, Heidelberg (2007)
33. Cortier, V., Delaune, S., Lafourcade, P.: A survey of algebraic properties used in cryptographic protocols. Journal of Computer Security 14(1), 1–43 (2006)
34. Cortier, V., Rusinowitch, M., Zalinescu, E.: A resolution strategy for verifying cryptographic protocols with CBC encryption and blind signatures. In: 7th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP) (2005)
35. Delaune, S.: Easy intruder deduction problems with homomorphisms. Information Processing Letters 97(6), 213–218 (2006)
36. Delaune, S., Kremer, S., Ryan, M.D.: Coercion-resistance and receipt-freeness in electronic voting. In: Proceedings of the 19th Computer Security Foundations Workshop (CSFW). IEEE Computer Society Press, Los Alamitos (2006)
37. Delaune, S., Lafourcade, P., Lugiez, D., Treinen, R.: Symbolic protocol analysis for monoidal equational theories. Information and Computation 206, 312–351 (2008)
38. Delaune, S., Lin, H.: Protocol verification via rigid/flexible resolution. In: Dershowitz, N., Voronkov, A. (eds.) LPAR 2007. LNCS (LNAI), vol. 4790. Springer, Heidelberg (2007)
39. Fermüller, C., Leitsch, A., Hustadt, U., Tamet, T.: Resolution decision procedure. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, ch. 25, vol. 2, pp. 1793–1849. North-Holland, Amsterdam (2001)
40. Huttel, H.: Deciding framed bisimulation. In: 4th International Workshop on Verification of Infinite State Systems INFINITY 2002, pp. 1–20 (2002)
41. Küsters, R., Tuengerthal, M.: Joint state theorems for public-key encryption and digital signature functionalities with local computations. In: Computer Security Foundations (CSF 2008) (2008)
42. Lowe, G.: An attack on the Needham-Schroeder public-key authentication protocol. Information Processing Letters 56(3), 131–133 (1996)
43. Lowe, G.: Towards a completeness result for model checking of security protocols. Journal of Computer Security 7(1) (1999)
44. McAllester, D.: Automatic recognition of tractability in inference relations. J. ACM 40(2) (1993)
45. Meadows, C.: The NRL protocol analyzer: An overview. Journal of Logic Programming 26(2), 113–131 (1996)
46. Millen, J., Shmatikov, V.: Symbolic protocol analysis with an Abelian group operator or Diffie-Hellman exponentiation. J. Computer Security (2005)
47. Millen, J.K., Ko, H.-P.: Narrowing terminates for encryption. In: Proc. Ninth IEEE Computer Security Foundations Workshop (CSFW) (1996)

48. Ramanujam, R., Suresh, S.P.: Tagging makes secrecy decidable with unbounded nonces as well. In: Pandya, P.K., Radhakrishnan, J. (eds.) FSTTCS 2003. LNCS, vol. 2914, pp. 363–374. Springer, Heidelberg (2003)
49. Rusinowitch, M., Turuani, M.: Protocol insecurity with finite number of sessions is np-complete. In: Proc. 14th IEEE Computer Security Foundations Workshop, Cape Breton, Nova Scotia (June 2001)
50. Weidenbach, C.: Towards an automatic analysis of security protocols in first-order logic. In: Ganzinger, H. (ed.) CADE 1999. LNCS (LNAI), vol. 1632, pp. 314–328. Springer, Heidelberg (1999)

Deciding Effectively Propositional Logic Using DPLL and Substitution Sets

Leonardo de Moura and Nikolaj Bjørner

Microsoft Research, One Microsoft Way, Redmond, WA, 98074, USA
{leonardo, nbjorner}@microsoft.com

Abstract. We introduce a DPLL calculus that is a decision procedure for the Bernays-Schönfinkel class, also known as EPR. Our calculus allows combining techniques for efficient propositional search with data-structures, such as Binary Decision Diagrams, that can efficiently and succinctly encode finite sets of substitutions and operations on these. In the calculus, clauses comprise of a sequence of literals together with a finite set of substitutions; truth assignments are also represented using substitution sets. The calculus works directly at the level of sets, and admits performing simultaneous constraint propagation and decisions, resulting in potentially exponential speedups over existing approaches.

1 Introduction

Effectively propositional logic, also known as the Bernays-Schönfinkel class, or EPR, of first-order formulas provides for an attractive generalization of pure propositional satisfiability and quantified Boolean formulas. The EPR class comprise of formulas of the form $\exists^*\forall^*\varphi$, where φ is a quantifier-free formula with relations, equality, but without function symbols. The satisfiability problem for EPR formulas can be reduced to SAT by first replacing all existential variables by skolem constants, and then grounding the universally quantified variables by all combinations of constants. This process produces a propositional formula that is exponentially larger than the original. In a matching bound, the satisfiability problem for EPR is NEXPTIME complete [1]. An advantage is that decision problems may be encoded exponentially more succinctly in EPR than with purely propositional encodings [2].

Our calculus aims at providing a bridge from efficient techniques used in pure SAT problems to take advantage of the succinctness provided for by the EPR fragment. One inspiration was [3], which uses an ad-hoc extension of a SAT solver for problems that can otherwise be encoded in QBF or EPR; and we hope the presented framework allows formulating such applications as strategies. A main ingredient is the use of sets of instantiations for both clauses and literal assignments. By restricting sets of instantiations to the EPR fragment, it is feasible to represent these using succinct data-structures, such as Binary Decision Diagrams [4]. Such representations allow delaying, and in several instances, avoiding, space overhead that a direct propositional encoding would entail.

The main contributions of this paper comprise of a calculus DPLL($\mathcal{SX}$) with substitution sets which is complete for EPR (Section 2). The standard calculus for propositional satisfiability lifts directly to DPLL($\mathcal{SX}$), allowing techniques from SAT solving to apply on purely propositional clauses. However, here, we will be mainly interested in investigating features specific to non-propositional cases. We make explicit and essential use of *factoring*, well-known from first-order resolution, but to our knowledge so far only used for clause simplifications in other liftings of DPLL. We show how the calculus lends itself to an efficient search strategy based on a simultaneous version of Boolean constraint propagation (Section 3). We exhibit cases where the simultaneous technique may produce an exponential speedup during propagation. By focusing on *sets of* substitutions, rather than substitutions, we open up the calculus to efficient implementations based on data-structures that can encode finite sets succinctly. Our current prototype uses a BDD package for encoding finite domains, and we report on a promising, albeit preliminary, empirical evaluation (Section 4). Section 5 concludes with related work and future extensions.

2 The DPLL($\mathcal{SX}$) Calculus

2.1 Preliminaries

We use $a, b, c, \triangle, \star, 0, \ldots$ to range over a finite alphabet Σ of constants, while $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}$ are tuples of constants, $x, y, z, x_0, x_1, x_2, \ldots$ for variables from a set $\mathcal{V}$, $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$ are tuples of variables, and $p_1, p_2, p, q, r, s, t, \ldots$ for atomic predicates of varying arities. Signed predicate symbols are identified by the set $\mathcal{L}$. As usual, literals (identified by the letter ℓ) are either atomic predicates or their negations applied to arguments. For example, $\overline{p}(x_1, x_2)$ is the literal where the binary atomic predicate p is negated. Clauses consist of a finite set of literals, where each atomic predicate is applied to distinct variables. For example $p(x_1, x_2) \vee \overline{q}(x_3) \vee \overline{q}(x_4)$ is a (well formed) clause. We use C, C', C_1, C_2 to range over clauses. The empty clause is identified by a $\Box$. Substitutions, written θ, θ', are idempotent partial functions from $\mathcal{V}$ to $\mathcal{V} \cup \Sigma$. In other words, $\theta \in \mathcal{V} \overset{\sim}{\to} \mathcal{V} \cup \Sigma$ is a substitution if for every $x \in \mathbf{Dom}(\theta)$ we have $\theta(x) \in \mathcal{V} \Rightarrow \theta(x) \in \mathbf{Dom}(\theta) \wedge \theta(\theta(x)) = \theta(x)$. Substitutions are lifted to homomorphisms in the usual way, so that substitutions can be applied to arbitrary terms, e.g., $\theta(f(c, x)) = f(c, \theta(x))$. Substitutions that map to constants only (Σ) are called *instantiations*. We associate with each substitution θ a set of instances, namely, $\mathit{instancesOf}(\theta) = \{\theta' \in (\mathbf{Dom}(\theta) \to \Sigma) \mid \forall x \in \mathbf{Dom}(\theta)\ .\ \theta'(x) = \theta'(\theta(x))\}$. Suppose θ is a substitution with domain $\{x_1, \ldots, x_n\}$; then the corresponding set $\mathit{instancesOf}(\theta)$ can be viewed as a set of n-entry records, or equivalently as an n-ary relation over $x_1, \ldots, x_n$. In the following, we can denote a set of substitutions as a set of instances, but will use the terminology *substitution set* to reflect that we use representations of such sets using a mixture of instantitions and proper substitutions. We discuss in more detail how substitution sets are represented in Section 3.3. As shorthand for *sets* of substitutions we use Θ, $\Theta', \Theta_1, \Theta_2$. A substitution-set constrained clause is a pair $C \cdot \Theta$ where C is a clause and Θ is a substitution set. The denotation of a constrained clause is the set of ground clauses $\theta(C)$, for every instantiation $\theta \in \Theta$. Notice that there are no ground clauses associated with $C \cdot \emptyset$, so our rules will avoid adding such tautologies. We will assume clauses are in "normal" form where the literals in C are applied to different variables, so it is up to the substitutions to create equalities between variables.

Literals can also be constrained, and we use the notation $\ell\Theta$ for the literal ℓ whose instances are determined by the non-empty Θ. If Θ is a singleton set, we may just write the instance of the literal directly. So for example $p(a)$ is shorthand for $p(x)\{a\}$.

Example 1 (Constrained clauses). The set of (unit) clauses: $p(a, b), p(b, c), p(c, d)$ can be represented as the set-constrained clause

$$p(x_1, x_2) \cdot \{[x_1 \mapsto a, x_2 \mapsto b], [x_1 \mapsto b, x_2 \mapsto c], [x_1 \mapsto c, x_2 \mapsto d]\},$$

or simply: $p(x_1, x_2) \cdot \{(a, b), (b, c), (c, d)\}$

A context Γ is a sequence of constrained literals and decision markers ($\diamond$). For instance, the sequence $\ell_1\Theta_1, \diamond, \ell_2\Theta_2, \ell_3\Theta_3, \ldots, \ell_k\Theta_k$ is a context. We allow concatenating contexts, so $\Gamma, \Gamma', \ell\Theta, \diamond, \ell'\Theta', \Gamma''$ is a context that starts with a sequence of constrained literals in Γ, continues with another sequence Γ', contains $\ell\Theta$, a decision marker, then $\ell'\Theta'$, and ends with Γ''.

Operations from relational algebra will be useful in manipulating substitution sets. See also [5], [6] and [7]. We summarize the operations we will be using below:

Selection $\sigma_{\varphi(\boldsymbol{x})}\Theta$ is shorthand for $\{\theta \in \Theta \mid \varphi(\theta(\boldsymbol{x}))\}$.

Projection $\pi_{\boldsymbol{x}}\Theta$ is shorthand for the set of substitutions obtained from Θ by removing domain elements other than $\boldsymbol{x}$. For example, $\pi_x\{[x \mapsto a, y \mapsto b], [x \mapsto a, y \mapsto c]\} = \{[x \mapsto a]\}$.

Co Projection $\hat{\pi}_{\boldsymbol{x}}\Theta$ is shorthand for the set of substitutions obtained from Θ by removing $\boldsymbol{x}$. So $\hat{\pi}_x\{[x \mapsto a, y \mapsto b], [x \mapsto a, y \mapsto c]\} = \{[y \mapsto b], [y \mapsto c]\}$.

Join $\Theta \bowtie \Theta'$ is the natural join of two relations. If Θ uses the variables $\boldsymbol{x}$ and $\boldsymbol{y}$, and Θ' uses variables $\boldsymbol{x}$ and $\boldsymbol{z}$, where $\boldsymbol{y}$ and $\boldsymbol{z}$ are disjoint, then $\Theta \bowtie \Theta'$ uses $\boldsymbol{x}, \boldsymbol{y}$ and $\boldsymbol{z}$ and is equal to $\{\theta \mid \hat{\pi}_{\boldsymbol{z}}(\theta) \in \Theta, \hat{\pi}_{\boldsymbol{y}}(\theta) \in \Theta'\}$. For example, $\{[x \mapsto a, y \mapsto b], [x \mapsto a, y \mapsto c]\} \bowtie \{[y \mapsto b, z \mapsto b], [y \mapsto b, z \mapsto a]\} = \{[x \mapsto a, y \mapsto b, z \mapsto b], [x \mapsto a, y \mapsto b, z \mapsto a]\}$.

Renaming $\delta_{\boldsymbol{x} \to \boldsymbol{y}}\Theta$ is the relation obtained from Θ by renaming the variables $\boldsymbol{x}$ to $\boldsymbol{y}$. We here assume that $\boldsymbol{y}$ is not used in Θ already.

Substitution $\theta(\Theta)$ applies the substitution θ to the set Θ. It is shorthand for a sequence of selection and co-projections. For example $[x \mapsto a](\Theta) = \hat{\pi}_x\sigma_{x=a}\Theta$.

Set operations $\Theta \cup \Theta'$ creates the union of Θ and Θ', both sets of n-ary tuples (sets of instances with n variables in the domain). Subtraction $\Theta \setminus \Theta'$ is the set $\{\theta \in \Theta \mid \theta \notin \Theta'\}$. The complement $\overline{\Theta}$ is $\Sigma^n \setminus \Theta$.

Notice, that we can compute the most general unifier of two substitutions θ and θ', by taking the natural join of their substitution set equivalents (the join is the empty set if the most general unifier does not exist). If two clauses $C \vee \ell(\boldsymbol{x})$ and $C' \vee \neg\ell(\boldsymbol{x})$, have substitution sets Θ and Θ' respectively, we can compute the resolvent $(C \vee C') \cdot \hat{\pi}_x(\Theta \bowtie \Theta')$. We quietly assumed that the variables in C and C' were disjoint and renamed apart from $\boldsymbol{x}$, and we will in the following assume that variables are by default assumed disjoint, or use appropriate renaming silently instead of cluttering the notation.

2.2 Inference Rules

We will adapt the proof calculus for DPLL($\mathcal{SX}$) from an exposition of DPLL($\mathcal{T}$) as an abstract transition system [8,9]. States of the transition system are of the form $\Gamma \,\|\, F$, where Γ is a context, and the set F is a collection of constrained clauses. Constrained literals are furthermore annotated with optional *explanations*. In this presentation we will use one kind of explanation of the form:

$\ell^{C \cdot \Theta}\Theta'$ All Θ' instances of ℓ are implied by propagation from $C \cdot \Theta$

We maintain the following invariant, which is crucial for conflict resolution:

Invariant 1. *For every derived context of the form* $\Gamma, \ell^{C \cdot \Theta}\Theta', \Gamma'$ *it is the case that* $C = (\ell_1 \vee \ldots \vee \ell_k \vee \ell(\boldsymbol{x}))$ *and there are assignments* $\overline{\ell}_i\Theta_i \in \Gamma$*, such that* $\Theta' \subseteq \hat{\pi}_{\boldsymbol{x}}(\Theta \bowtie \delta_{\boldsymbol{x} \to \boldsymbol{x}_1}\Theta_1 \bowtie \ldots \bowtie \delta_{\boldsymbol{x} \to \boldsymbol{x}_k}\Theta_k)$ *(written* $\Theta' \subseteq \hat{\pi}_{\boldsymbol{x}}(\Theta \bowtie \Theta_1 \bowtie \ldots \bowtie \Theta_k)$ *without the explicit renamings). Furthermore,* Θ' *is non-empty.*

We take advantage of this invariant to associate a function *premises*$(\ell^{C \cdot \Theta}\Theta')$ that extracts $\Theta \bowtie \Theta_1 \bowtie \ldots \bowtie \Theta_k$ (or, to spell it out for the last time: $\Theta \bowtie \delta_{\boldsymbol{x} \to \boldsymbol{x}_1}\Theta_1 \bowtie \ldots \bowtie \delta_{\boldsymbol{x} \to \boldsymbol{x}_k}\Theta_k$).

During conflict resolution, we also use states of the form

$$\Gamma \,\|\, F \,\|\, C \cdot \Theta, \Theta_r$$

where $C \cdot \Theta$ is a conflict clause, and Θ_r is the set of instantiations that falsify C; it is used to guide conflict resolution.

We split the presentation of DPLL($\mathcal{SX}$) into two parts, consisting of the search inference rules, given in Fig. 1, and the conflict resolution rules, given in Fig. 2. Search proceeds similarly to propositional DPLL: It starts with the initial state $\|\, F$ where the context is empty. The result is either unsat indicating that F is unsatisfiable, or a state $\Gamma \,\|\, F$ such that every clause in F is satisfied by Γ. A sequence of Decide steps are used to guess an assignment of truth values to the literals in the clauses F. The side-conditions for UnitPropagate and Conflict are similar: they check that there is a non-empty join of the clause's substitutions with substitutions associated with the complemented literals. There is a conflict when all of the literals in a clause has complementary assignments, otherwise, if all but one literal has a complementary assignment, we may apply unit propagation to the remaining literal. The last side condition on UnitPropagate prevents it from being applied if there is a conflict. Semantically, a non-empty join implies that there are instances of the literal assignments that contradict (all but one of) the clause's literals.

$$\mathsf{Decide}\ \frac{\ell \in F \quad \text{If } \overline{\ell}\Theta' \in \Gamma \text{ or } \ell\Theta' \in \Gamma\text{, then } \Theta \bowtie \Theta' = \emptyset}{\Gamma \parallel F \implies \Gamma, \diamond, \ell\Theta \parallel F}$$

$$\mathsf{Conflict}\ \frac{C = (\ell_1 \vee \ldots \vee \ell_k),\ \overline{\ell}_i\Theta_i \in \Gamma, \quad \Theta_r = \Theta \bowtie \Theta_1 \bowtie \ldots \bowtie \Theta_k \neq \emptyset}{\Gamma \parallel F, C \cdot \Theta \implies \Gamma \parallel F, C \cdot \Theta \parallel C \cdot \Theta, \Theta_r}$$

$$\mathsf{UnitPropagate}\ \frac{\begin{array}{l} C = (\ell_1 \vee \ldots \vee \ell_k \vee \ell(\boldsymbol{x})),\ \overline{\ell}_i\Theta_i \in \Gamma, i = 1, .., k \\ \Theta' = \pi_{\boldsymbol{x}}(\Theta \bowtie \Theta_1 \bowtie \ldots \bowtie \Theta_k) \setminus \bigcup\{\Theta_\ell \mid \ell\Theta_\ell \in \Gamma\} \neq \emptyset \\ \Theta' \bowtie \bigcup\{\Theta_{\overline{\ell}} \mid \overline{\ell}\Theta_{\overline{\ell}} \in \Gamma\} = \emptyset \end{array}}{\Gamma \parallel F, C \cdot \Theta \implies \Gamma, \ell^{C \cdot \Theta} \cdot \Theta' \parallel F, C \cdot \Theta}$$

Fig. 1. Search inference rules

$$\mathsf{Resolve}\ \frac{\begin{array}{c} \pi_{\boldsymbol{y}}\Theta_r \bowtie \Theta_\ell = \emptyset \ \text{ for every } \ell(\boldsymbol{y}) \in C',\ C_\ell = (C(\boldsymbol{y}) \vee \overline{\ell}(\boldsymbol{x})) \\ \Theta_r' = \hat{\pi}_{\boldsymbol{x}}(\Theta_r \bowtie \Theta_\ell \bowtie \mathit{premises}(\overline{\ell}\Theta_\ell)) \neq \emptyset, \quad \Theta'' = \hat{\pi}_{\boldsymbol{x}}(\Theta \bowtie \Theta') \end{array}}{\Gamma, \overline{\ell}^{C_\ell \cdot \Theta'}\Theta_\ell \parallel F \parallel (C'(\boldsymbol{z}) \vee \ell(\boldsymbol{x})) \cdot \Theta, \Theta_r \implies \Gamma \parallel F \parallel (C(\boldsymbol{y}) \vee C'(\boldsymbol{z})) \cdot \Theta'', \Theta_r'}$$

$$\mathsf{Skip}\ \frac{\pi_{\boldsymbol{y}}\Theta_r \bowtie \Theta_\ell = \emptyset \ \text{ for every } \ell(\boldsymbol{y}) \in C}{\Gamma, \overline{\ell}^{C_\ell \cdot \Theta'}\Theta_\ell \parallel F \parallel C \cdot \Theta, \Theta_r \implies \Gamma \parallel F \parallel C \cdot \Theta, \Theta_r}$$

$$\mathsf{Factoring}\ \frac{\Theta_r' = \hat{\pi}_{\boldsymbol{z}}\sigma_{\boldsymbol{y}=\boldsymbol{z}}\Theta_r \neq \emptyset, \quad \Theta' = \hat{\pi}_{\boldsymbol{z}}\sigma_{\boldsymbol{y}=\boldsymbol{z}}\Theta}{\Gamma \parallel F \parallel (C(\boldsymbol{x}) \vee \ell(\boldsymbol{y}) \vee \ell(\boldsymbol{z})) \cdot \Theta, \Theta_r \implies \Gamma \parallel F \parallel (C(\boldsymbol{x}) \vee \ell(\boldsymbol{y})) \cdot \Theta', \Theta_r'}$$

$$\mathsf{Learn}\ \frac{C \cdot \Theta \notin F}{\Gamma \parallel F \parallel C \cdot \Theta, \Theta_r \implies \Gamma \parallel F, C \cdot \Theta \parallel C \cdot \Theta, \Theta_r}$$

$$\mathsf{Unsat}\ \frac{\Theta \neq \emptyset}{\Gamma \parallel F \parallel \Box \cdot \Theta, \Theta_r \implies \mathsf{unsat}}$$

$$\mathsf{Backjump}\ \frac{\begin{array}{l} C = (\ell_1 \vee \ldots \vee \ell_k \vee \ell(\boldsymbol{x})),\quad \overline{\ell}_i\Theta_i \in \Gamma_1 \\ \Theta' = \pi_{\boldsymbol{x}}(\Theta \bowtie \Theta_1 \bowtie \ldots \bowtie \Theta_k) \setminus \bigcup\{\Theta_\ell \mid \ell\Theta_\ell \in \Gamma_1\} \neq \emptyset \end{array}}{\Gamma_1, \diamond, \Gamma_2 \parallel F \parallel C \cdot \Theta, \Theta_r \implies \Gamma_1, \ell^{C \cdot \Theta}\Theta' \parallel F}$$

$$\mathsf{Refine}\ \frac{\emptyset \neq \Theta_1' \subset \Theta_1}{\Gamma, \diamond, \ell\Theta_1, \Gamma' \parallel F \parallel C \cdot \Theta, \Theta_r \implies \Gamma, \diamond, \ell\Theta_1' \parallel F}$$

Fig. 2. Conflict resolution rules

Conflict resolution rules, shown in Fig. 2, produce resolution proof steps based on a clause identified in a conflict. The Resolve rule unfolds literals from conflict clauses that were produced by unit propagation, and Skip bypasses propagations that were not used in the conflict. The precondition of Resolve only applies if there is a single literal that is implied by the top of the context, if that is not the case, we can use factoring. Unlike propositional DPLL, it is not always possible to apply factoring on repeated literals in a conflict clause. We therefore include a Factoring rule to explicitly handle factoring when it applies. Any clause derived by resolution or factoring can be learned using Learn, and added to the clauses in F.

The inference system produces the result unsat if conflict resolution results in the empty clause. There are two ways to transition from conflict resolution to search mode. Back-jumping applies when all but one literal in the conflict clause is assigned below the current decision level. In this case the rule $\mathsf{Backjump}$ adds the uniquely implied literal to the logical context Γ and resumes search mode. As factoring does not necessarily always apply, we need another rule, called Refine, for resuming search. The Refine rule allows refining the set of substitutions applied to a decision literal. The side condition to Refine only requires that Θ_1 be a non-empty, non-singleton set. In some cases we can use the conflict clause to guide refinement: If C contains two occurrences $\overline{\ell}(\boldsymbol{x}_1)$ and $\overline{\ell}(\boldsymbol{x}_2)$, where $\pi_{\boldsymbol{x}_1}(\Theta_r)$ and $\pi_{\boldsymbol{x}_2}(\Theta_r)$ are disjoint but are subsets of Θ_1, then use one of the projections as Θ_1'.

To illustrate the inference rules, and in particular the use of factoring and splitting consider the following example. Assume we have the clauses:

$$F : \begin{cases} C_1 : \overline{p}(x) \;\vee\; q(y) \;\vee\; \overline{r}(z) \cdot \{(a,a,a)\}, \\ C_2 : \overline{p}(x) \;\vee\; s(y) \;\vee\; \overline{t}(z) \cdot \{(b,b,b)\}, \\ C_3 : \overline{q}(x) \;\vee\; \overline{s}(y) \cdot \{(a,b)\} \end{cases}$$

A possible derivation may start with the empty assignment and take the shape shown in Fig 3. We end up with a conflict clause with two occurrences of $\overline{p}$.

$$\begin{aligned}
& \parallel F \\
\Longrightarrow\ & \mathsf{Decide} \\
& \diamond, r(x)\{a,b,c\} \parallel F \\
\Longrightarrow\ & \mathsf{Decide} \\
& \diamond, r(x)\{a,b,c\}, \diamond, t(x)\{b,c\} \parallel F \\
\Longrightarrow\ & \mathsf{Decide} \\
& \diamond, r(x)\{a,b,c\}, \diamond, t(x)\{b,c\}, \diamond, p(x)\{a,b\} \parallel F \\
\Longrightarrow\ & \mathsf{UnitPropagate\ using}\ C_1 \\
& \diamond, r(x)\{a,b,c\}, \diamond, t(x)\{b,c\}, \diamond, p(x)\{a,b\}, q(x)\{a\} \parallel F \\
\Longrightarrow\ & \mathsf{UnitPropagate\ using}\ C_2 \\
& \underbrace{\diamond, r(x)\{a,b,c\}, \diamond, t(x)\{b,c\}, \diamond, p(x)\{a,b\}, q(x)\{a\}, s(x)\{b\}}_{\Gamma} \parallel F \\
\Longrightarrow\ & \mathsf{Conflict\ using}\ C_3 \\
& \Gamma \parallel F \parallel \overline{q}(x) \;\vee\; \overline{s}(y) \cdot \{(a,b)\} \\
\Longrightarrow\ & \mathsf{Resolve\ using}\ C_2 \\
& \Gamma \parallel F \parallel \overline{p}(x) \vee \overline{q}(y) \vee \overline{t}(z) \cdot \{(b,a,b)\} \\
\Longrightarrow\ & \mathsf{Resolve\ using}\ C_1 \\
& \Gamma \parallel F \parallel \overline{p}(x) \vee \overline{p}(x') \vee \overline{r}(y) \vee \overline{t}(z) \cdot \{(b,a,a,b)\}
\end{aligned}$$

Fig. 3. Example derivation steps

Factoring does not apply, because the bindings for x and x' are different. Instead, we can choose one of the bindings as the new assignment for p. For example, we could take the subset $\{b\}$, in which case the new stack is:

$$\begin{aligned}
\Longrightarrow\ & \mathsf{Refine} \\
& \diamond, r(x)\{a,b,c\}, \diamond, t(x)\{b,c\}, \diamond, p(x)\{b\} \parallel F
\end{aligned}$$

2.3 Soundness, Completeness and Complexity

It can be verified by inspecting each rule that all clauses added by conflict resolution are resolvents of the original set of clauses F. Thus,

Theorem 1 (Soundness). *DPLL($\mathcal{SX}$) is sound.*

The use of premises and the auxiliary substitution Θ_r have been delicately formulated to ensure that conflict resolution is finite and stuck-free, that is, Resolve and Factoring always produce a conflict clause that admits either Backjump or Refine.

Theorem 2 (Stuck-freeness). *For every derivation starting with rule* Conflict *there is a state $\Gamma \parallel F \parallel C \cdot \Theta, \Theta_r$, such that* Backjump *or* Refine *is enabled.*

Note that Refine is always enabled when there is a non-singleton set attached to a decision literal, but the key property that is relevant for this theorem is that adding *premises* to a resolvent does not change the projection of old literals in the resolved clause. In other words, for every $\boldsymbol{y}$ disjoint from $premises(\overline{\ell}\Theta_\ell)$ it is the case that: $\pi_{\boldsymbol{y}}(\Theta_r) = \pi_{\boldsymbol{y}}(\Theta_r \bowtie premises(\overline{\ell}\Theta_\ell))$.
Similarly, we can directly simulate propositional grounding in the calculus, so:

Theorem 3 (Completeness). *DPLL($\mathcal{SX}$) is complete for EPR.*

The calculus admits the expected asymptotic complexity of EPR. Suppose the maximal arity of any relation is a, the number of constants is $n = |\Sigma|$, and the number of relations is m, set $K \leftarrow m \times (n^a)$, then:

Theorem 4 (Complexity). *The rules of DPLL($\mathcal{SX}$) terminate with at most $O(K \cdot 2^K)$ applications, and with maximal space usage $O(K^2)$.*

Proof. First note that a context can enumerate each literal assignment explicitly using at most $O(K)$ space, since each of the m literals should be evaluated at up to n^a instances. Literals that are tagged by explanations require up to additional $O(K)$ space, each.

For the number of rule applications, consider the ordering $\prec$ on contexts defined as the transitive closure of:

$$\Gamma, \ell'\Theta', \Gamma' \prec \Gamma, \diamond, \ell\Theta, \Gamma'' \tag{1}$$
$$\Gamma, \diamond, \ell\Theta', \Gamma' \prec \Gamma, \diamond, \ell\Theta, \Gamma'' \quad \text{when } \Theta' \subset \Theta \tag{2}$$

The two rules for $\prec$ correspond to the inference rules Backjump and Refine that generate contexts of decreased measure with respect to $\prec$. Furthermore, we may restrict our attention to contexts Γ where for every literal ℓ, such that $\Gamma = \Gamma', \ell\Theta, \Gamma''$ if $\exists\Theta' \,.\, \ell\Theta' \in \Gamma', \Gamma''$, then $\Theta' \bowtie \Theta = \emptyset$. There are at most $K!$ such contexts, but we claim the longest $\prec$ chain is at most $K \cdot 2^K$. First, if all sets are singletons, then the derivation is isomorphic to a system with K atoms, which requires at most 2^K applications of the rule (1). Derivations that use non-singleton sets embed directly into a derivation with singletons using more

$$\textsf{S-UnitPropagate}\ \frac{\begin{array}{l} C = (\ell_1 \vee \ldots \vee \ell_k \vee \ell(\boldsymbol{x})), \\ \Theta'_i = \cup\{\Theta_i \mid \overline{\ell}_i\Theta_i \in \Gamma\},\ \Theta'_\ell = \bigcup\{\Theta_\ell \mid \ell\Theta_\ell \in \Gamma\}, \\ \Theta' = \pi_{\boldsymbol{x}}(\Theta \bowtie \Theta'_1 \bowtie \ldots \bowtie \Theta'_k) \setminus \Theta'_\ell \neq \emptyset \end{array}}{\Gamma \parallel F, C \cdot \Theta \implies \Gamma, \ell^{C\cdot\Theta} \cdot \Theta' \parallel F, C \cdot \Theta}$$

$$\textsf{S-Conflict}\ \frac{\begin{array}{l} C = (\ell_1 \vee \ldots \vee \ell_k),\ \Theta'_i = \cup\{\Theta_i \mid \overline{\ell}_i\Theta_i \in \Gamma\}, \\ \Theta' = \Theta \bowtie \Theta'_1 \bowtie \ldots \bowtie \Theta'_k \neq \emptyset \end{array}}{\Gamma \parallel F, C \cdot \Theta \implies \Gamma \parallel F, C \cdot \Theta \parallel C \cdot \Theta, \Theta'}$$

Fig. 4. S-UnitPropagate and S-Conflict

steps. Finally, rule (2) may be applied at most K times between each step that corresponds to a step of the first kind.

Note that the number of rule applications does not include the cost of manipulating substitution sets. This cost depends on the set representations. While the asymptotic time complexity is (of course) no better than what a brute force grounding provides, DPLL($\mathcal{SX}$) only really requires space for representing the logical context Γ. While the size of Γ may be in the order K, there is no requirement for increasing F from its original size. As we will see, the use of substitution sets may furthermore compress the size of Γ well below the bound of K. One may worry that in an implementation, the overhead of using substitution sets may be prohibitive compared to an approach based on substitutions alone. Section 3.3 describes a data-structure that compresses substitution sets when they can be represented as substitutions directly.

3 Refinements of DPLL($\mathcal{SX}$)

The calculus presented in Section 2 is a general framework for using substitution sets in the context of DPLL. We first discuss a refinement of the calculus that allows to apply unit propagation for several assignments simultaneously. Second, we examine data-structures and algorithms for representing, indexing and manipulating substitution sets efficiently during search.

3.1 Simultaneous Propagation and FUIP-Based Conflict Resolution

In general, literals are assigned substitutions at different levels in the search. Unit propagation and conflict detection can therefore potentially be identified based on several different instances of the same literals. For example, given the clause $(\overline{p}(x) \vee q(x))$ and the context $p(a), p(b)$, unit propagation may be applied on $p(a)$ to imply $q(a)$, but also on $p(b)$ to imply $q(b)$. We can factor such operations into simultaneous versions of the UnitPropagate and Conflict rules. The simultaneous version of the propagation and conflict rules take the form shown in Fig 4.

Correctness of these simultaneous versions rely on the basic the property that $\bowtie$ distributes over unions:

$$(R \cup R') \bowtie Q \;=\; (R \bowtie Q) \cup (R' \bowtie Q) \;\text{ for every } R, R', Q \tag{3}$$

Thus, every instance of S-UnitPropagate corresponds to a set of instances of UnitPropagate, and for every S-Conflict there is a selection of literals in Γ that produces a Conflict. The rules suggest to maintain accumulated sets of substitutions per literal, and apply propagation and conflict detection rules once per literal, as opposed to once per literal occurrence in Γ. A trade-off is that we break invariant 1 when using these rules. Instead we have:

Invariant 2. *For every derived context of the form* $\Gamma, \ell^{C \cdot \Theta}\Theta', \Gamma'$ *where* $C = (\ell_1 \vee \ldots \vee \ell_k \vee \ell)$, *it is the case that* $\Theta' \subseteq \hat{\pi}_{\boldsymbol{x}}(\Theta \bowtie \Theta'_1 \bowtie \ldots \bowtie \Theta'_k)$ *where* $\Theta'_i = \bigcup\{\Theta_i \mid \overline{\ell}_i\Theta_i \in \Gamma\}$.

The weaker invariant still suffices to establish the theorems from Section 2.3. The function *premises* is still admissible, thanks to (3).

Succinctness. Note the asymmetry between the use of simultaneous unit propagation and the conflict resolution strategy: while the simultaneous rules allow to use literal assignments from several levels at once, conflict resolution traces back the origins of the propagations that closed the branches. The net effect may be a conflict clause that is exponentially larger than the depth of the branch. As an illustration of this situation consider the clauses where p is an n-ary predicate:

$$\begin{array}{l} \neg p(0,\ldots,0) \;\wedge\; shape(\star) \;\wedge\; shape(\triangle) \\ \wedge_i \, [p(\boldsymbol{x}, \star, 0, .., 0) \wedge p(\boldsymbol{x}, \triangle, 0, .., 0) \;\rightarrow\; p(\boldsymbol{x}, 0, 0.., 0)] \text{ where } \boldsymbol{x} = x_0, \ldots, x_{i-1} \\ \wedge_{0 \le j < n} shape(x_j) \;\rightarrow\; p(x_0, \ldots, x_{n-1}) \end{array} \tag{4}$$

Claim. The clauses are contradictory, and any resolution proof requires 2^n steps.

Justification. Backchaining from $p(0,\ldots,0)$, we observe that all possible derivations are of the form:

$$\begin{array}{rl} p(0,\ldots,0) & \leftarrow p(\star,0,\ldots,0), p(\triangle,0,\ldots,0) \\ & \leftarrow p(\star,\triangle,0,..,0), p(\star,\star,0,..,0), p(\triangle,\triangle,0,..,0), p(\triangle,\star,0,..,0) \\ & \leftarrow \ldots \\ & \leftarrow p(\star,\star,\ldots),\ldots,p(\triangle,\triangle,\ldots) \text{ all } 2^n \text{ combinations} \\ & \leftarrow shape(\star)\ldots shape(\triangle) \end{array}$$

Claim. DPLL($\mathcal{SX}$) with simultaneous unit propagation requires $O(n)$ steps to complete the derivation.

Justification. The two assertions $shape(\star)$ and $shape(\triangle)$ may be combined into $shape(x)\{\star,\triangle\}$ and then used to infer $p(\boldsymbol{x})\{\star,\triangle\} \times \ldots \times \{\star,\triangle\}$ in one propagation. Each consecutive propagation may be used to produce $p(\boldsymbol{x})\Theta$, where Θ contains a suffix with k consecutive 0's and the rest being all combinations of $\star$ and $\triangle$.

In this example, we did in fact not need to perform conflict resolution at all because the problem was purely Horn, and no decisions were required to derive the empty clause. But it is simple to modify such instances to non-Horn problems, and the general question remains how and whether to avoid an exponential cost of conflict resolution as measured by the number of propagation steps used to derive the conflict.

One crude approach for handling this situation is to abandon conflict resolution if the size of the conflict clause exceeds a threshold. When abandoning conflict resolution apply Refine, which is enabled as long as there is at least one decision literal on the stack whose substitution set is a non-singleton. If all decision literals use singletons, then Refine does not apply. In this case we have to use a different way of backtracking. We can flip the last decision literal under the context of the negation of all decision literals on the stack. The rule corresponding to this approach is U(nit)-Refine:

$$\text{U-Refine}\ \frac{\begin{array}{c}\boldsymbol{c} \in \Sigma^k \quad \diamond \notin \Gamma', \quad \ell_1\Theta_1, \ldots, \ell_m\Theta_m \text{ are the decision literals in } \Gamma \\ C' = \overline{\ell} \vee \overline{\ell}_1 \vee \ldots \vee \overline{\ell}_m, \Theta' = \{\boldsymbol{c}\} \bowtie \Theta_1 \bowtie \ldots \bowtie \Theta_m \text{ is a singleton}\end{array}}{\Gamma, \diamond, \ell\{\boldsymbol{c}\}, \Gamma' \parallel F \parallel C \cdot \Theta, \Theta_r \implies \Gamma, \overline{\ell}\{\boldsymbol{c}\}^{C' \cdot \Theta'} \parallel F}$$

But with succinct substitution sets it is sometimes possible to match the succinctness of unit propagation during conflict resolution. The approach we are going to present will use ground clauses during resolution. The ground clauses can have multiple occurrences of the same predicate symbol, but applied to different (ground) arguments. We then summarize the different arguments using a substitution set, so that the representation of the ground clause only requires each predicate symbol at most twice (positive and/or negated), but with a potentially succinct representation of the arguments.

Suppose S-Conflict infers the conflict clause $C \cdot \Theta$ and set Θ_r. Let θ_0 be an arbitrary instantiation in Θ_r. Initialize the map Ψ from the set of signed predicate symbols to substitution sets as follows:

$$\Psi(\ell) \leftarrow \bigcup \{\pi_{\boldsymbol{x}_i}\theta_0 \mid \ell(\boldsymbol{x}_i) \in C\}, \quad \text{for } \ \ell \in \mathcal{L}. \tag{5}$$

Note that a clause C may have multiple occurrences of a predicate symbol with the same sign, but applied to different arguments. The definition ensures that if $\ell \in \mathcal{L}$ is a signed predicate symbol that does not occur in C, then $\Psi(\ell) = \emptyset$.

Example 2. Assume $C = p(x_1) \vee p(x_2) \vee q(x_3)$ and $\theta = (a, b, c)$, then $\Psi(p) = \{a, b\}, \Psi(\overline{p}) = \emptyset, \Psi(q) = \{c\}, \Psi(\overline{q}) = \emptyset$.

We can directly reconstruct a clause from Ψ by creating a disjunction of $\Sigma_{\ell \in \mathcal{L}} |\Psi(\ell)|$ literals and a substitution that is the product of all elements in the range of Ψ. This inverse mapping is called *clause_of*(Ψ). Sets in the range of Ψ may get large, but we can here rely on the same representation as used for substitution sets. We can now define a (first-unique implication point) resolution strategy that works using Ψ. We formulate the strategy separately from the already introduced rules, as we need to ensure that we can maintain the representation of the conflict clause using Ψ.

$$\boxed{\begin{array}{l}
\mathit{resolve}(\Gamma_1, \diamond, \Gamma_2, \Psi) = \mathsf{Backjump} \text{ with } \Gamma_1 \ell^{C\cdot\Theta}\{\boldsymbol{c}\} \text{ if} \\
\qquad \ell \in \mathbf{Dom}(\Psi),\ \boldsymbol{c} \in \Psi(\ell), \quad \Psi(\ell) \setminus \{\boldsymbol{c}\} \subseteq \bigcup\{\Theta' \mid \overline{\ell}\Theta' \in \Gamma_1\} \\
\qquad \Psi(\ell') \subseteq \bigcup\{\Theta' \mid \overline{\ell'}\Theta' \in \Gamma_1\} \text{ for } \ell \neq \ell' \\
\qquad C \cdot \Theta = \mathit{clause\_of}(\Psi) \\
\mathit{resolve}(\Gamma, \ell\Theta, \Psi) = \mathit{resolve}(\Gamma, \Psi) \text{ if } \Theta \cap \Psi(\ell) = \emptyset \\
\mathit{resolve}(\Gamma, \ell^{C\vee\ell\cdot\Theta}\Theta', \Psi) = \mathit{resolve}(\Gamma, \Psi), \text{ if } \Theta' \cap \Psi(\ell) = \{\boldsymbol{c}\}, \text{ and where} \\
\quad \Psi(\ell) \leftarrow \Psi(\ell) \setminus \Theta' \\
\quad \text{for } \ell'(\boldsymbol{x}) \in C\text{: } \Psi(\ell') \leftarrow \Psi(\ell') \cup \pi_{\boldsymbol{x}}(\mathit{premises}(\ell^{C\vee\ell\cdot\Theta}\Theta')) \\
\mathit{resolve}(\Gamma, \diamond, \ell\Theta, \Psi) = \mathsf{Refine} \text{ if other rules don't apply.}
\end{array}}$$

Besides the cost of performing the set operations, the strategy still suffers from the potential of generating an exponentially large implied learned clause $C \cdot \Theta'$ during backjumping. An implementation can choose to resort to applying Refine or U-Refine in these cases. We do not have experimental experience with the representation in terms of Ψ. Instead our prototype implementation uses Refine together with U-Refine, when the conflict resolution steps start generating conflict clauses with more than 20 (an arbitrary default) literals.

3.2 Selecting Decision Literals and Substitution Sets

Selecting literals and substitution sets blindly for Decide is possible, but not a practical heuristic. As in the Model-evolution calculus [10], we take advantage of the current assignment Γ to guide selection. Closure of substitution sets under complementation streamlines the task a bit for the case of DPLL($\mathcal{SX}$). First observe that Γ induces a default interpretation of the instances of every atom p by taking:

$$[\![p]\!] = \bigcup\{\Theta' \mid p\Theta' \in \Gamma\} \quad \text{and} \quad [\![\overline{p}]\!] = \overline{[\![p]\!]} \tag{6}$$

Note that we can assume that Γ is consistent, so $\bigcup\{\Theta' \mid \overline{p}\Theta' \in \Gamma\} \subseteq [\![\overline{p}]\!]$. Using the current assignment for the positive literals and the complement thereof for negative ones is an arbitrary choice in the context of DPLL($\mathcal{SX}$). One may fix a default interpretation differently for each atom. But note that this particular choice coincides with negation as failure for the case of Horn clauses.

We now say that ℓ_i is a candidate decision literal with instantiation Θ'_i if there is a clause $C \cdot \Theta$, such that $C = (\ell_1 \vee \ldots \vee \ell_k)$, $1 \leq i \leq k$, and:

$$\Theta'_i = (\Theta \bowtie [\![\overline{\ell}_1]\!] \bowtie \ldots \bowtie [\![\overline{\ell}_k]\!]) \setminus \bigcup\{\Theta' \mid \overline{\ell}_i\Theta' \in \Gamma\} \neq \emptyset \tag{7}$$

Our prototype uses a greedy approach for selecting decision literals and substitution sets: predicates with lower arity are preferred over predicates with higher arities. In particular, propositional atoms are used first and they are assigned using standard SAT heuristics. Predicates with non-zero arity that are not completely assigned are checked for condition (7) and we pick the first applicable candidate. The process either produces a decision literal, or determines that the current set of clauses are satisfiable in the default interpretation, as the following easy lemma summarizes:

Lemma 1. *If for a state $\Gamma \parallel F, (\ell_1 \vee \ldots \vee \ell_k) \cdot \Theta$ it is the case that neither* UnitPropagate *or* Conflict *are enabled and*

$$\Theta \bowtie [\![\overline{\ell}_1]\!] \bowtie \ldots \bowtie [\![\overline{\ell}_k]\!] \neq \emptyset \tag{8}$$

then there is some i, such that $1 \leq i \leq k$, that satisfies (7). Furthermore, the identified substitution Θ'_i is disjoint from any Θ', where $\ell_i\Theta' \in \Gamma$ or $\overline{\ell}_i\Theta' \in \Gamma$. Conversely, if the current state is closed under propagation and conflict and there is no clause that satisfies (8), then the default interpretation is a model for the set of clauses F.

Proof. If the current state is closed under Conflict and UnitPropagate, then for every clause $(\ell_1 \vee \ldots \vee \ell_k) \cdot \Theta$: $\Theta \bowtie \Theta'_1 \bowtie \ldots \bowtie \Theta'_k = \emptyset$ for $\Theta'_i = \bigcup\{\Theta' \mid \overline{\ell}_i\Theta' \in \Gamma\}$. Suppose that (8) holds, then by $\bigcup\{\Theta' \mid \overline{p}\Theta' \in \Gamma\} \subseteq [\![\overline{p}]\!]$ and distributivity of $\bowtie$ over $\cup$ there is some i where (7) holds. The converse direction is immediate.

3.3 Hybrid Substitution Sets

Representing all substitution sets directly as BDDs is not practical. In particular, computing $\Theta \bowtie \Theta'_1 \bowtie \ldots \bowtie \Theta'_k$ by directly applying the definitions of $\bowtie$ as conjunction and $\delta_{\rightarrow}$ as BDD renaming does not work in practice for clauses with several literals: simply building a BDD for Θ (where Θ is the substitution set associated with a clause) can be prohibitively expensive. We here investigate a representation of substitution sets called *hybrid substitution sets* that admit pre-compiling and factoring several of the operations used during constraint propagation. The format is furthermore amenable to a two-literal watch strategy for the propositional case.

Definition 1 (Hybrid substitution sets). *A hybrid substitution set is a pair (θ, Θ), where θ is a substitution, and Θ is a relation (substitution set). Furthermore, the domain of Θ consists of the variables where θ is idempotent. That is, $\mathbf{Dom}(\Theta) = \{x \in \mathbf{Dom}(\theta) \mid \theta(x) = x\}$. The substitution set associated with a hybrid substitution is the set of instances: $\Theta \bowtie \{\theta\}$.*

In one extreme, a proper substitution θ is equivalent to the hybrid substitution set $(\theta, \top)$. Our prototype compiles clauses into such substitution sets. In the other extreme, every substitution set Θ can be represented as (id, Θ), where id is the identity substitution over the domain of Θ. Our prototype uses such substitution sets to constrain literals. We will henceforth take the liberty to abuse notation and treat substitutions θ and substitution sets Θ also as hybrid substitution sets.

Hybrid substitution sets are attractive because common operations are cheap (linear time) when the substitutions are proper. They also enjoy closure properties under the main relational algebraic operations that are used in conflict resolution.

Lemma 2. *Proper substitutions are closed under the operations:* $\bowtie$, $\pi_{\boldsymbol{x}}$, $\hat{\pi}_{\boldsymbol{x}}$, $\sigma_{\boldsymbol{x}=\boldsymbol{y}}$, *and* $\delta_{\boldsymbol{x}\to\boldsymbol{y}}$, *but not under union nor complementation.*

Even if the hybrid substitution sets are not proper, the complexity of the common operations is reduced by using the substitution component when it is not the identity. For example, representing each variable in a BDD requires $\log(|\Sigma|)$ bits, and if the bits of two variables are spaced apart by k other bits, the operation that restricts a BDD equating the two variables may cause a size increase of up to 2^k. The problem can be partially addressed using static or dynamic variable reordering techniques, but variable orderings have to be managed carefully when variables are shared among several substitution sets.

Constraint Propagation. Consider a clause $C \cdot (\theta, \Theta) \in F$ and a substitution Θ'_i (associated with literal $\overline{\ell}_i(\boldsymbol{x})$ in Γ, where ℓ_i occurs C). The main operation during constraint propagation is computing $(\theta, \Theta) \bowtie \delta_{\boldsymbol{x}\to\boldsymbol{x}_i}\Theta'_i$, which is equivalent to

$$(\theta, \Theta \bowtie r_i(\Theta'_i)) \quad \text{where} \quad r_i = [x \mapsto \theta(\delta_{\boldsymbol{x}\to\boldsymbol{x}_i} x) \mid x \in \boldsymbol{x}]. \tag{9}$$

The equivalence suggests to pre-compute and store the substitution r_i, for every clause C and literal in C. Each renaming may be associated with several clauses; and we can generalize the two-literal watch heuristic for a clause C by using watch literals ℓ_i and ℓ_j from C as guards if the current assignments Θ'_i and Θ'_j satisfy $r_i(\Theta'_i) = r_j(\Theta'_j) = \emptyset$.

Resolution. In general, when taking the join of two hybrid substitution sets we have the equivalence: $(\theta, \Theta) \bowtie (\theta', \Theta') = (m, m(\Theta) \bowtie m(\Theta'))$, where $m = mgu(\theta, \theta')$, if the most general unifier exists, otherwise the join is $(id, \bot)$. Resolution requires computing $\hat{\pi}_{\boldsymbol{x}}((\theta, \Theta) \bowtie (\theta', \Theta'))$ or in general $\hat{\pi}_{\boldsymbol{x}}(\delta_{y\to z}(\theta, \Theta) \bowtie \delta_{u\to v}(\theta', \Theta'))$ where $\boldsymbol{x}, y, z, u, v$ are suitable vectors of variables. Again, we can compose the re-namings first with the substitutions, compute the most general unifier $m = mgu(\delta_{y\to z}\theta, \delta_{u\to v}\theta')$, in such a way that if $m(y) = m(x)$, for $x \in \boldsymbol{x}, y \notin \boldsymbol{x}$, then $m(y)$ is a constant or maps to some variable also not in $\boldsymbol{x}$; and returning $(m \setminus \boldsymbol{x}, \exists \boldsymbol{x}(m(\Theta) \wedge m(\Theta')))$. It is common for BDD packages to supply a single operation for $\exists \boldsymbol{x}(\varphi \wedge \psi)$.

4 Implementation and Evaluation

We implemented DPLL($\mathcal{SX}$) as a modification of the propositional SAT solver used in the SMT solver Z3. The implementation associates with each clause a hybrid substitution set and pre-compiles the set of substitutions r_i used in (9). This allows the BDD package, we use BuDDy[1], to cache results from repeated substitutions of the same BDDs (the corresponding operation is called `vec_compose`

[1] `http://buddy.wiki.sourceforge.net`

in BuDDy). BDD caching was more generally useful in obliterating special purpose memoization in the SAT solver. For instance, we attempted to memoize the default interpretations of clauses as they could potentially be re-used after back-tracking, but we found so far no benefits of this added memoization over relying on the BDD cache. BuDDy supports finite domains directly making it easier to map a problem with a set of constants $\Sigma = c_1, \ldots, c_k$ into a finite domain of size $2^{\lceil \log(k) \rceil}$. Rounding the domain size up to the nearest power of 2 does not change satisfiability of the problem, but has a significant impact on the performance of BDD operations. Unfortunately, we have not been able to get dynamic variable re-ordering to work with finite domains in BuDDy, so all our results are based on a fixed default variable order.

As expected, our prototype scales reasonably well on formula (4). It requires n propagations to solve an instance where p has arity n. With $n = 10$ takes $0.01s.$, $n = 20$ takes $0.2s.$, and $n = 200$ takes $18s.$ (and caches 1.5M BDD nodes, on a 32bit, 2GHz, 2GB, TS2500). Darwin [11] handles $n = 10$ in 0.4 seconds and 2049 propagations, while increasing n to 20 is already too overwhelming.

Example 3. Suppose p is an n-ary predicate, and that we have n unary predicates $a_0, \ldots, a_{n-1}$, then consider the (non-Horn) formula:

$$\begin{aligned}&\wedge_{0 \le i < n} \forall \boldsymbol{x} \ . \ [p(\boldsymbol{x}) \rightarrow p(.., x_{i-1}, 1, x_{i+1}, ..)] \\ &\wedge\ p(0, \ldots, 0) \ \wedge \ \wedge_{0 \le i < n} (a_i(0) \vee a_i(1)) \ \wedge \ \forall \boldsymbol{x} \ . \ [(\wedge_i a_i(x_i)) \rightarrow \neg p(\boldsymbol{x})]\end{aligned} \tag{10}$$

DPLL($\mathcal{SX}$) uses n simultaneous propagations to learn the assignment $p(\boldsymbol{x})\top$. In contrast, standard unit propagation requires 2^n steps. Since no splitting was required to learn this assignment, it can be used to eliminate p from conflict clauses during lemma learning. The resulting conflict clauses during backjumping are then $\forall \boldsymbol{x} \ . \ \bigvee_{0 \le i < m} \neg a_i(x_i)$ for $m = n-1, \ldots 1$. Accordingly, the prototype uses 0.06 seconds for $n = 30$, $0.9s$ for $n = 80$, and 26s. for $n = 200$, while even a very good instantiation based prover Darwin requires $O(2^{n-1})$ branches, which is reflected in the timings: for $n =$ 11, 12, 13, 14, 15, 16, take $1, 4, 16, 60, 180, 477$ seconds respectively.

We also ran our prototype on the CASC-21 benchmarks from the EPS and EPT divisions. In the EPT division fails to prove `PUZ037-3.p`, with a timeout of 120 seconds, as the BDDs built during propagation blow upIt solves the other 49 problems, using less than 1 second for all but `SYN439-1.p`, which requires 894 conflicts and 9.8 seconds. In the EPS division our prototype solves 46 out of 50 problems within the given 120s. timeout.

5 Conclusions

Related Work DPLL($\mathcal{SX}$) is a so called *instance*-based method [12] and it shares several features with instance-based implementations derived from DPLL, such as the Model Evolution Calculus ($\mathcal{ME}$) calculus [10], the iProver [13], and the earlier work on a primal-dual approach for satisfiability of EPR [14]. These methods are also decision procedures for EPR that go well beyond direct propositional grounding (as do resolution methods [15]). Lemma learning in $\mathcal{ME}$ [11] comprises of two rules GRegress and a non-ground lifting Regress. In a somewhat rough analogy to Regress, the resolution rules used in DPLL($\mathcal{SX}$) uses the set Θ_r to guide a more general lifting for the produced conflict clause. Connections between relational technology and theorem proving were made in [5], as well as [6]. The use of BDDs for compactly representing relations is wide-spread. Of high relevance to DPLL($\mathcal{SX}$) is the system BDDBDDB, which is a Datalog engine based on BDDs [7]. Semantics of negation in Datalog aside, DPLL($\mathcal{SX}$) essentially reduces to BDDBDDB for Horn problems. Simultaneous unit propagation is for instance implicit in the way clauses get compiled to predicate transformers, but on the other hand, apparatus for handling non-Horn problems is obviously absent from BDDBDDB.

Extensions. A number of compelling extensions to DPLL($\mathcal{SX}$) remain to be investigated. For example, we may merge two clauses $C{\cdot}\Theta$ and $C{\cdot}\Theta'$ by taking the union of the substitution sets. The clause $C \cdot \Theta'$ could for instance be obtained by resolving binary clauses, so this feature could simulate iterative squaring known from symbolic model checking. Examples where iterative squaring pays of are provided in [16]. We currently handle equality in our prototype by supplying explicit equality axioms (reflexivity, symmetry, transitivity, and congruence) for the binary equality relation $\simeq$, but supporting equality as an intrinsic theory is possible and the benefits would be interesting to study. We have also to work out efficient ways of building in subsumption. Supporting other theories is also possible by propagating all instances from substitution sets, but it would be appealing to identify cases where an explicit enumeration of substitution sets can be avoided. We used reduced ordered BDDs in our evaluation of the calculus, but this is by no means the only possible representation. We may for instance delay forming canonical decision diagrams until it is required for evaluating (non-emptiness) queries (a technique used for Boolean Expression Diagrams). It would also be illustrative to investigate how DPLL($\mathcal{SX}$) applies to finite model finding and general first-order problems. Darwin(FM) already addressed using EPR for finite model finding, and as GEO [17] exemplifies, one can extend finite model finders to the general first-order setting.

Another avenue to pursue is relating our procedure with methods used for QBF. While there is a more or less direct embedding of QBF into EPR (obtained by Skolemization) the decision problem for QBF is *only* PSPACE complete, while the procedure we outlined requires up to exponential space.

Thanks to the referees for their very detailed and constructive feedback on the submitted version of this paper.

References

1. Lewis, H.R.: Complexity results for classes of quantificational formulas. J. Comput. Syst. Sci. 21, 317–353 (1980)
2. Pérez, J.A.N., Voronkov, A.: Encodings of Bounded LTL Model Checking in Effectively Propositional Logic. In: [18], pp. 346–361.
3. Dershowitz, N., Hanna, Z., Katz, J.: Bounded Model Checking with QBF. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 408–414. Springer, Heidelberg (2005)
4. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Transactions on Computers C-35, 677–691 (1986)
5. Voronkov, A.: Merging relational database technology with constraint technology. In: Bjorner, D., Broy, M., Pottosin, I.V. (eds.) PSI 1996. LNCS, vol. 1181, pp. 409–419. Springer, Heidelberg (1996)
6. Tammet, T., Kadarpik, V.: Combining an inference engine with database: A rule server. In: Schröder, M., Wagner, G. (eds.) RuleML 2003. LNCS, vol. 2876, pp. 136–149. Springer, Heidelberg (2003)
7. Whaley, J., Avots, D., Carbin, M., Lam, M.S.: Using datalog with binary decision diagrams for program analysis. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 97–118. Springer, Heidelberg (2005)
8. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). J. ACM 53, 937–977 (2006)
9. Krstic, S., Goel, A.: Architecting Solvers for SAT Modulo Theories: Nelson-Oppen with DPLL. In: Konev, B., Wolter, F. (eds.) FroCos 2007. LNCS (LNAI), vol. 4720, pp. 1–27. Springer, Heidelberg (2007)
10. Baumgartner, P., Tinelli, C.: The model evolution calculus as a first-order DPLL method. Artif. Intell. 172, 591–632 (2008)
11. Baumgartner, P., Fuchs, A., Tinelli, C.: Lemma learning in the model evolution calculus. In: Hermann, M., Voronkov, A. (eds.) LPAR 2006. LNCS (LNAI), vol. 4246, pp. 572–586. Springer, Heidelberg (2006)
12. Baumgartner, P.: Logical engineering with instance-based methods. In: [18], pp. 404–409.
13. Ganzinger, H., Korovin, K.: New directions in instantiation-based theorem proving. In: LICS, pp. 55–64. IEEE Computer Society Press, Los Alamitos (2003)
14. Gallo, G., Rago, G.: The satisfiability problem for the Schönfinkel-Bernays fragment: partial instantiation and hypergraph algorithms. Technical Report 4/94, Dip. Informatica, Universit'a di Pisa (1994)
15. Fermüller, C.G., Leitsch, A., Hustadt, U., Tammet, T.: Resolution decision procedures. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning, pp. 1791–1849. Elsevier and MIT Press (2001)
16. Pérez, J.A.N., Voronkov, A.: Proof systems for effectively propositional logic. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008 (2008)
17. de Nivelle, H., Meng, J.: Geometric resolution: A proof procedure based on finite model search. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 303–317. Springer, Heidelberg (2006)
18. Pfenning, F. (ed.): CADE 2007. LNCS (LNAI), vol. 4603. Springer, Heidelberg (2007)

Proof Systems for Effectively Propositional Logic

Juan Antonio Navarro[1] and Andrei Voronkov[2]

[1] Max Planck Institute for Software Systems
[2] The University of Manchester

Abstract. We consider proof systems for effectively propositional logic. First, we show that propositional resolution for effectively propositional logic may have exponentially longer refutations than resolution for this logic. This shows that methods based on ground instantiation may be weaker than non-ground methods. Second, we introduce a generalisation rule for effectively propositional logic and show that resolution for this logic may have exponentially longer proofs than resolution with generalisation. We also discuss some related questions, such as sort assignments for generalisation.

1 Introduction

Effectively propositional logic (in the sequel we call it simply EPR) is a fragment of first-order logic which can be effectively translated into propositional logic. Formulae in EPR are, essentially, those formulae in the Bernays-Schönfinkel class. It has recently been shown that several real life applications such as bounded model checking [11] and planning [12] can be naturally and succinctly encoded as EPR formulae. Effectively propositional benchmarks in the TPTP library [19] also include problems from diverse areas such as algebra, natural language processing, verification and puzzles.

When skolemised, EPR formulae contain no function symbols and thus have a finite Herbrand Universe, which allows one to translate them to propositional logic using *grounding*: substitutions of constants for variables of the formula.

Grounding and the subsequent use of SAT solvers remains one of the most succesful approaches to checking the satisfiability of EPR formulae. The purpose of this paper is to compare several proof systems for EPR formulae, including those based on grounding and SAT. Although our results are formulated in terms of proof lengths, which is not always the most interesting criterion in practice, they give some insight on why proof systems for EPR can be more powerful than propositional proof systems. We believe that the insight gained from our results and their proofs may find their ways in practical proof systems for EPR formulae. In particular, we propose a new inference rule, called *generalisation*, which allows one to lift ground (propositional) reasoning to the non-propositional level.

Our first result is that on EPR formulae resolution can be exponentially more efficient than any propositional proof system working on a set of ground instances of EPR formulae. Although this is not surprising, Plaisted and Zhu [15] have indeed came across a similar result comparing the power of variants of resolution, we include here a simple proof in our context of interest. The proof is done by giving a family of formulae which have first-order resolution refutations of quadratic size, but whose propositional refutations are all exponential in size.

Our second result is that resolution with generalisation can have exponentially shorter proofs than resolution. Moreover, we define variants of the generalisation rule based on sort inference methods which allow one to generalise from a smaller set of clauses. We then show that generalisation using sorts is sound. We also show that some standard sort inference algorithms, such as the one implemented in the Paradox model finder [3], are compatible with generalisation.

2 Preliminaries

A clause is called an *EPR clause* if it has no function symbols of arity greater than 0. That is, EPR clauses may contain variables and constants but no function symbols. In this paper we will consider only EPR clauses. An expression (for example, a term, a literal or a clause) is called *ground* if it contains no variables. An *instance* of a clause C is any clause $C\sigma$ obtained from C by applying a substitution σ.

Given a set of clauses S, we denote by S^* the set of all ground instances of clauses in S:

$$S^* \overset{\text{def}}{=} \{S\sigma \mid S\sigma \text{ is ground and } S\sigma \text{ contains only constants from } S\} .$$

As usual, if S contains no constant symbols, then an arbitrary constant is added in order to compute S^*. We consider logic without equality, which is not a real restriction for EPR formulae since equality can be axiomatised using EPR formulae. Note that a collection S of ground EPR clauses can also be considered as a collection of propositional clauses by taking every ground atom A ocurriong in S as a distinct propositional variable.

To improve readability, we will sometimes write clauses as implications, for example $p \wedge q \rightarrow r$ instead of $\neg p \vee \neg q \vee r$.

2.1 Propositional vs First-Order Resolution

We prove in this section that resolution on EPR formulae can be exponentially more efficient than propositional proof systems using grounding. First, we should explain what we mean by the latter. By a propositional proof system for EPR clauses we mean any proof system that, given a set S of clauses, builds a subset of S^* and applies a propositional proof system to it. Note that if a propositional proof system finds an unsatisfiable subset of S^*, then S is unsatisfiable too.

We will now proceed to show that there is a family of sets of clauses S_i with respective resolution refutations Γ_i, such that the shortest propositional refutation of S_i is exponentially larger than Γ_i. The following construction, and a similar theorem, can be found in the work of Plaisted and Zhu [15] (Theorem 2.4.11 in pp. 89-90) where the power of variants of first-order resolution are compared.

Although our result follows from some remarks made in the proof of that theorem, for completeness and clarity we include here again both the construction and the full proof in the context of comparing the power of propositional versus effectively propositional resolution.

Let i be a positive integer. Consider the language having two constant symbols 0 and 1, and a single predicate symbol s of arity i. Denote by $\bar{0}$, $\bar{1}$, $\bar{x}$, etc. sequences of constants 0, 1 and variables, respectively, whose length will be clear from the context. The set S_i consists of the following: the clause

$$s(\bar{0}) \; , \tag{1}$$

the clause

$$\neg s(\bar{1}) \; , \tag{2}$$

and i clauses of the form:

$$s(\bar{x}, 0, \bar{1}) \rightarrow s(\bar{x}, 1, \bar{0}) \; , \tag{3}$$

where the length of $\bar{1}$ ranges from $0, \ldots, i-1$. Note that the size of S_i is $O(i^2)$.

Lemma 1. *The shortest propositional refutation of S_i has a size of $O(2^i)$.*

Proof. Note that every ground atom in the language of S_i is of the form $s(\bar{b})$, where $\bar{b}$ is a sequence of bits of length i. We can consider this sequence of bits as a non-negative integer between 0 and $2^i - 1$ written in binary notation. For a number n such that $0 \leq n < 2^i$ let us denote by $\underline{n}$ the sequence of i bits corresponding to that number. Using this notation we can observe that S_i^* consists of the following clauses: $s(\underline{0})$, $\neg s(\underline{2^i - 1})$, and all clauses of the form $s(\underline{n}) \rightarrow s(\underline{n+1})$ for $n = 0, \ldots, 2^i - 1$.

Moreover, it is not hard to argue that every proper subset of S_i^* is satisfiable, so every propositional refutation of S_i should be at least as large as S_i^*. It remains to note that the number of literals in S_i^* is $O(2^i)$. □

Lemma 2. *S_i has a resolution refutation of the size $O(i^2)$.*

Proof. Let us build, for every $k = 1, \ldots, i-1$ a resolution derivation Π_k from S_i of the clause

$$s(\bar{x}, \bar{0}) \rightarrow s(\bar{x}, \bar{1}), \tag{4}$$

where k is the length of $\bar{0}$. This derivation will be built by induction on k. When $k = 1$, (4) is of the form (3). For $k > 1$, we will take the derivation Π_{k-1} and add several clauses to it to obtain Π_k. We know that Π_{k-1} derives a clause

$$s(\bar{x}, y, \bar{0}) \rightarrow s(\bar{x}, y, \bar{1}) \; . \tag{5}$$

where the length of $\bar{0}$ is k. From this and (3) we derive by a resolution inference the clause

$$s(\bar{x}, 0, \bar{0}) \rightarrow s(\bar{x}, 1, \bar{0}) \; .$$

From this and (5) we derive by a resolution inference the clause

$$s(\bar{x}, 0, \bar{0}) \rightarrow s(\bar{x}, 1, \bar{1}) \ .$$

and we are done. It is easy to see that the number of clauses in Π_k not ocurring in S is $3(k-1)$, so the size of Π_k is $O(i \cdot k)$.

Note that Π_i derives

$$s(\bar{0}) \rightarrow s(\bar{1})$$

and has the size $O(i^2)$. From this clause, (1) and (2) we can derive the empty clause in 2 steps, so there exists a resolution refutation of S_i having the size $O(i^2)$. $\square$

Lemmas 1 and 2 yield the following theorem

Theorem 1. *There is a sequence of sets of clauses $S_1, S_2, \ldots$ of increasing size such that each S_i has a refutation of a size quadratic in i, while the shortest propositional refutation of S_i has a size exponential in i.*

Interestingly, as the results of the recent CASC competition show [18], resolution has so far been found not very competitive on EPR formulae. In the following section we introduce another inference rule, especially designed for effectively propositional formulae and intended to complement the resolution approach.

3 The Generalisation Inference Rule

Suppose that S is a set of EPR clauses whose constants are in the set $\{c_1, \ldots, c_n\}$. Let $A[x]$ be a quantifier-free formula with a free variable x written in the same language as S. Let us note that the formula

$$\forall x(A[c_1] \wedge \cdots \wedge A[c_n] \rightarrow A[x]) \qquad (6)$$

is valid in all Herbrand models of S. This, and the fact that $A[x]$ is quantifier-free, implies that adding (6) to S does not change the set of Herbrand models. This gives us an idea of a *generalization rule*: an inference rule that allows one to derive $A[x]$ from $A[c_1], \ldots, A[c_n]$. We will introduce a more general form of this rule inspired by grounding systems based on sort inference [see e.g. 17]. These systems may derive that the set of instantiations for some variable in s may be restricted to a subset of all constants. Likewise, we can make generalisation from a subset of constants.

Let us give some definitions. We call a *predicate position* a pair (p, i) where p is a predicate symbol and i is a positive integer less than or equal to the arity of p. When we denote positions we will write $p.i$ instead of (p, i). We call a *sort* a set of constant symbols, and a *sort assignment* a function that maps each predicate position to a sort. A ground formula F is said to be *compatible with a sort assignment* A if for every atomic subformula $p(t_1, \ldots, t_n)$ of F, we have $t_i \in A(p.i)$. For a quantifier-free formula F, denote by $F|_A$ the set of all ground instances of F compatible with A.

Definition 1. *Given a sort assignment A, the inference rule of* generalisation with respect to A *is*

$$\frac{C_1 \vee p(\bar{t}_1, c_1, \bar{u}_1) \quad \cdots \quad C_n \vee p(\bar{t}_n, c_n, \bar{u}_n)}{C_1\sigma \vee \cdots \vee C_n\sigma \vee p(\bar{t}_1, y, \bar{u}_1)\sigma}\ \mathsf{Gen}_A$$

where y is a fresh variable, the length of $\bar{t}_i$ is k, $A(p.k{+}1) = \{c_1, \ldots, c_n\}$ and σ is the most general simultaneous unifier of the set of tuples $\{(\bar{t}_1, \bar{u}_1), \ldots, (\bar{t}_n, \bar{u}_n)\}$.

If both inference rules, resolution and generalization, are allowed then the system is (trivially) complete; this follows since resolution alone is already complete. The discussion in the beginning of this section also suggests that it is also sound when the sort function always returns the set of all constants.

One of the first results that we want to show, is that when resolution is combined with generalisation, then the obtained inference system is still, as resolution alone, refutationally sound and complete. For this we begin by introducing the following set of clauses which will be useful to simulate generalisation using only resolution.

Definition 2. *Given a sort assignment A, we define the set of* generalisation clauses*, denoted by $[\![A]\!]$, as the set containing all the clauses of the form*

$$p(\bar{x}, c_1, \bar{z}) \wedge \cdots \wedge p(\bar{x}, c_n, \bar{z}) \rightarrow p(\bar{x}, y, \bar{z}) \ , \tag{7}$$

where k is the length of $\bar{x}$ and $A(p.k+1) = \{c_1, \ldots, c_n\}$.

Notice that all clauses in $[\![A]\!]|_A$ are tautologies, since the variable y would have to be mapped to a constant c_i of its appropriate sort.

Lemma 3. *If there is a refutation of a set of clauses S using resolution and generalisation with respect to a sort assignment A, then there is a refutation of $S \cup [\![A]\!]$ using only resolution.*

Proof. The result easily follows by noticing that generalisation inference steps can be simulated by resolving the n clauses of the form $C_i \vee s(\bar{x}, c_i, \bar{z})\sigma_i$ with the corresponding clause in the set $[\![A]\!]$ from Definition 2. □

Let us now discuss sort inference functions that result in sort assignments preserving satisfiability.

Definition 3. *A* sort inference function Ξ *is a function that yields a sort assignment given a set of clauses as input. Moreover, we say that Ξ is*

– valid *if, for any set of clauses S, the sets S and $S|_A$ are equisatisfiable; and*
– stable *if, for any set of clauses S, $\Xi(S \cup [\![A]\!]) = A$.*

where $A = \Xi(S)$.

The first condition, validity, states that when checking the satisfiability of S by using instantiation-based methods, the generation of ground instances can be restricted to those which are compatible with A. The condition of stability asserts that the sort inference procedure is not affected when the set of generalisation clauses $[\![A]\!]$ is added to a formula.

The following theorem proves that resolution can be extended with generalisation preserving soundness.

Theorem 2. *Let S be a set of clauses, Ξ be a valid and stable sort inference function, and $A = \Xi(S)$. If there is a refutation of S using resolution and generalisation with respect to A, then there is a refutation of S using resolution only.*

Proof. We will first show that the set of clauses S is equisatisfiable with $S \cup [\![A]\!]$. Since the sort inference function is stable, we know that $\Xi(S) = \Xi(S \cup [\![A]\!]) = A$. Moreover, since the sort inference is valid, the sets $S \cup [\![A]\!]$ and $(S \cup [\![A]\!])|_A = S|_A \cup [\![A]\!]|_A$ are equisatisfiable. But recall that $[\![A]\!]|_A$ contains only tautologies and, therefore, $S|_A \cup [\![A]\!]|_A$ and $S|_A$ are equisatisfiable. Finally, again by the validity of the sort inference function, we get that $S|_A$ and S are equisatisfiable.

Now, to prove the theorem statement, suppose that there is a refutation of S using resolution and generalisation with respect to A. Then, by Lemma 3 and since resolution is sound, $S \cup [\![A]\!]$ is unsatisfiable. But then, from the previous paragraph, it follows that S is unsatisfiable and, since resolution is refutation complete, there is a refutation of S using resolution only. $\square$

From this, our main result now follows as a simple corollary.

Corollary 1. *An inference rule system based on resolution and generalisation, with respect to a valid and stable sort inference function, is both refutationally sound and complete.*

Proof. Soundness follows by Theorem 1, while completeness is directly inherited from the completeness of resolution. $\square$

From this result it follows that, any valid and stable sort inference function is suitable to be combined with generalisation in order to produce a sound and complete inference system. The question on how to obtain such kind of sort inference functions, however, remains open. This is the matter of the following section.

3.1 Sort Inference for Generalisation

In this section we will explore some possibilities in order to generate sort information that can be combined with the generalisation inference rule. A first option, though not very interesting, is to assign the trivial sort assignment to all sets of clauses.

Definition 4. *Given an effectively propositional language with a domain $\mathcal{D}$ of constant symbols, the* trivial sort assignment *is the function that maps every predicate position to $\mathcal{D}$.*

That is, it uses the domain of the logic itself as the sort for all variables and positions in predicates. This procedure is clearly stable since, irrelevant to the particular set of input clauses, the trivial sort assignment is always used. Moreover, by Herbrand's theorem this procedure is also valid and therefore a suitable candidate to be used together with generalisation.

In the following Section 3.2, we will see how even this simple approach can already represent a significant advantage over using the resolution inference rule alone. However, particularly on problems from applications, it is very likely that more specialised sort inference functions are able to give even better results in practise.

An example of a sort inference function is the method proposed by Claessen and Sörensson [3] and implemented in Paradox in the context of grounding-based model finding.

Algorithm 1 (Basic sort inference). Start with a sort assignment giving unrelated empty sorts to each predicate position.

Processing one clause at a time, and as a union-find algorithm: for each variable in the clause, merge the sorts assigned to all predicate positions where that variable occurs; and, for each constant symbol, add the constant symbol to the sort assigned to the predicate position where it appears.

Finally, add a dummy constant symbol to any sort that still remained empty at the end of this procedure.

It is not hard to argue, as it is done by Claessen and Sörensson [3], that this sort inference function is valid —in the sense of our Definition 3— and that, moreover, is not affected when adding the set of clauses $[\![A]\!]$ to S. So, from Theorem 2, it follows that we already have a sort inference method that, without any further modifications, can be directly used to empower generalisation inferences.

It is to be expected, that if one obtains a sort inference method by extending some available technique to restrict the number of generated instances in a grounding approach, then the obtained method will most likely be valid. This follows since such methods actually work by replacing the satisfiability testing for an effectively propositional formula, to checking instead an equisatisfiable set of propositional instances. This equisatisfiability is, precisely, what the property of validity asks for.

Unfortunately, however, not any ground restriction method can be so easily integrated with generalisation. Using the idea of positional linking, also called structural constraints by Schulz [17] and implemented in eground, one can easily define the following sort inference function.

Algorithm 2 (Positional sort inference). Given a set of clauses S compute, for every signed predicate $s.i$, the set $T_{s.i}$ as the set of all terms that appear in a literal with a signed predicate s at position i, i.e. if $s(t_1, \ldots, t_n)$ appears in S

then $t_i \in T_{s.i}$. Then let $C_{s.i} = T_{s.i}$ if all terms in $T_{s.i}$ are constants, and $C_{s.i} = \mathcal{D}$ otherwise.

Now, for each predicate position $p.i$, the *positional sort inference* is defined as the function that maps each such set S, to the sort assignment $A_{p.i} = C_{p.i} \cap C_{\neg p.i}$.

This sort inference function is clearly also valid. It works by the observation that literals which are not compatible with the generated sort assignment would be pure, i.e. they only appear in one of the two possible phases, and so they can be discarded. However, this sort inference function is not stable, as the following example shows.

Example 1. Consider the following satisfiable set of clauses S:

$$p(a)$$
$$\neg p(x) \vee q(x)$$
$$\neg q(b)$$

The positional sort would have $A_{p.1} = \{a\}$, $A_{q.1} = \{b\}$, and the restricted set $S|_A$ is simply $\{p(a), \neg q(b)\}$. Note that, however, adding the clause

$$\neg p(a) \vee p(x)$$

which is part of $[\![A]\!]$, would cause $A_{p.1} = \mathcal{D} = \{a, b\}$ —because now a variable appears in $p(x)$ on both positive and negative phases— making the sort inference function unstable and rendering the set of clauses $S \cup [\![A]\!]$ unsatisfiable.

In this section we have shown how a sort inference method, as proposed by Claessen and Sörensson [3], can be used together with the generalisation inference rule in order to make it more easily applicable in practice. In the following we give, in the form of a theoretical result, some evidence on why combining generalisation with resolution is likely to produce a powerful reasoning system.

3.2 Generalisation vs Resolution

In this section, and in order to further motivate the use of the generalisation inference rule in combination with resolution, we show a family of formulae which, similar to the one given in Section 2.1, shows that refutations can become exponentially shorter when combining resolution with the generalisation inference rule. For doing so we will show an example of a series of unsatisfiable sets of clauses $S_1, \ldots, S_n$ such that the length of shortest resolution refutation of S_n is exponential in n, while using both generalisation and resolution it is possible to find a refutation of size quadratic in n. In the following we will use x_i to represent variables, b_i and c_i for constant symbols, as well as s_i and t_i for arbitrary terms.

Definition 5. *Take a logic whose language has a set of constant symbols* $\mathbb{B} = \{0, 1\}$ *and let* n *be a non-negative number. For every* i*, with* $0 \leq i \leq n$*, there is a pair of predicate symbols* p_i *and* q_i *both of arity* i*.*

Now let S_n be the set of clauses that contains: the clause

$$p_0 \ , \tag{8}$$

$2n$ clauses, two for every $0 \le i < n$, of the form

$$\begin{aligned} p_i(x_1, \ldots, x_i) &\rightarrow p_{i+1}(x_1, \ldots, x_i, 0) \ , \\ p_i(x_1, \ldots, x_i) &\rightarrow p_{i+1}(x_1, \ldots, x_i, 1) \ , \end{aligned} \tag{9}$$

the clause

$$p_n(x_1, \ldots, x_n) \rightarrow q_n(x_1, \ldots, x_n) \ , \tag{10}$$

n clauses, one for every $0 \le i < n$, of the form

$$q_{i+1}(0, x_{\underline{i}}, \ldots, x_n) \wedge q_{i+1}(1, x_{\underline{i}}, \ldots, x_n) \rightarrow q_i(x_{\underline{i}}, \ldots, x_n) \ , \tag{11}$$

where $\underline{i} = n - i + 1$, and the clause

$$\neg q_0 \ . \tag{12}$$

Moreover, we will assume that generalisation inferences are applied with respect to the trivial sort assignment that simply maps every predicate position to the domain set $\mathbb{B} = \{0, 1\}$.

Intuitively, clauses of the form (9) encode the fact that if $p_i(b_1, \ldots, b_i)$ is true, then p_n should be true for all n-bit strings with a prefix of $b_1, \ldots, b_i$. A dual of this is encoded by clauses of the form (11): if $q_i(b_{\underline{i}}, \ldots, b_n)$ is false, then q_n should be false for some n-bit string with a suffix of $b_{\underline{i}}, \ldots, b_n$.

From clauses (8) and (9) we get that $p_n(b_1, \ldots, b_n)$ is true for all n-bit strings. Then from (10) that $q_n(b_1, \ldots, b_n)$ is also true for all n-bit strings and, therefore from (11), the atom q_0 should be true. But this causes a contradiction with (12), so the set S_n is unsatisfiable.

Indices in variables and terms have been chosen to enforce the *prefix* and *suffix* intuition of these predicate symbols. Formally, in the atoms $p_i(t_1, \ldots, t_i)$ and $q_i(t_{\underline{i}}, \ldots, t_n)$, we say that the position of the term t_i is the i-th *bit position*. Note that in all clauses of S_n, the variable x_i only appears at the i-th bit position of an atom.

Theorem 3. *There is a refutation of S_n, using both generalisation and resolution inference rules, which is of size quadratic in n.*

Proof. We start our refutation with the clause (8) which is the fact p_0. Observe now that it is possible to extend a proof of

$$p_i(x_1, \ldots, x_i) \tag{13}$$

to a proof of

$$p_{i+1}(x_1, \ldots, x_i, x_{i+1}) \tag{14}$$

by adding a constant number of steps.

To do this, first apply a generalisation inference on the pair of clauses (9) to obtain

$$p_i(x_1, \ldots, x_i) \rightarrow p_{i+1}(x_1, \ldots, x_i, x_{i+1}) \ ,$$

and then resolve this with (13) to obtain (14).

After n iterations of this procedure we get a proof of $p_n(x_1, \ldots, x_n)$ whose length is linear in n. Now, resolve this with (10) to obtain $q_n(x_1, \ldots, x_n)$. Observe now that we can extend a proof of

$$q_{i+1}(x_{\underline{i}-1}, x_{\underline{i}}, \ldots, x_n) \tag{15}$$

to a proof of

$$q_i(x_{\underline{i}}, \ldots, x_n) \tag{16}$$

by adding a constant number of steps.

To do this, simply resolve (15) with (11) to obtain

$$q_{i+1}(1, x_{\underline{i}}, \ldots, x_n) \rightarrow q_i(x_{\underline{i}}, \ldots, x_n) \ ,$$

and again with (15) to finally obtain (16).

After n of such iterations we end with a proof of q_0 which is also of length linear in n. Finally resolving q_0 with (12) we obtain a refutation of S_n. Since the size of each clause in the refutation is also linear in n, the size of the refutation is quadratic in n. □

Our main theorem of this section states that, using resolution alone, even the shortest refutation is of length at least exponential in n. The idea of the proof is not very difficult, but proving some of the necessary lemmas gets rather involved. We will therefore give first a sketch of the proof, followed by a couple of lemmas without proofs, followed by a formal proof of the main theorem relying on those lemmas. The missing proofs, as well as some other aditional details, can be found in an appendix of the full version of this paper.[1]

Theorem 4. *A resolution refutation of S_n has a length of, at least, 2^n.*

Proof (Sketch of proof). To prove that any refutation of S_n has at least an exponential length, we introduce a function on sets of clauses that, in a way, measures the accumulated progress achieved step by step on a refutation.

This *work* function, denoted by w, will map the set of clauses occurring in a partial proof Γ to the set of n-bit strings which, intuitively, have already been *consumed* while trying to build a refutation. This function should moreover satisfy $w(S_n) = \emptyset$, while the work of any refutation Γ is $w(\Gamma) = \mathbb{B}^n$. In order to prove the theorem statement it would then be enough to show that from one step to the next in the refutation, the work done increases in, at most, a single element. In other words all elements in $\mathbb{B}^n$ would have to be consumed one by one, yielding a proof of exponential length. □

[1] Available at: `http://www.mpi-sws.mpg.de/~jnavarro/papers.html`

In order to define such work function, first we identify the kind of clauses that might appear in a refutation of S_n. We observe that, indeed, only two kinds of clauses are possible (more details in the paper's full version):

- type I are clauses, such as (8–10), of the form
$$[l_1] \rightarrow h \tag{17}$$
where the head h has a predicate symbol p_j, with $0 \leq j \leq n$, or q_n.
- type II are clauses, such as (11–12), of the form
$$l_1 \wedge \cdots \wedge l_m \rightarrow h \tag{18}$$
where the head h is either $\perp$ or has a predicate symbol q_i with $i < n$. Moreover, we say that a literal l_i is *active* if its bit positions overlap with those of h, and *inactive* otherwise.

It is easy to check that resolving together two clauses of type I will yield another clause of type I, while resolving a clause of type II with one of either type will also produce a clause of type II.

Definition 6 (Work function). *For a clause of type II with head $q_i(t_{\underline{i}}, \ldots, t_n)$ we first define the* work of a literal l *as follows:*

$$w(l) = \begin{cases} \mathbb{B}^n & l = \perp, \\ \mathbb{B}^{n-j} \times \hat{t}_{\underline{j}} \times \cdots \times \hat{t}_n & l = q_j(t_{\underline{j}}, \ldots, t_n), \\ \hat{t}_1 \times \cdots \times \hat{t}_n & l = p_j(t_1, \ldots, t_j) \textit{ for an active } l, \\ \emptyset & \textit{if } l \textit{ is inactive.} \end{cases}$$

where $\hat{t}$ is the set of all ground instances of the term t. We also let $w^c(l) = \mathbb{B}^n \setminus w(l)$. The work of the clause C *is then defined as*

$$w(C) = w^c(l_1) \cap \cdots \cap w^c(l_m) \cap w(h) \tag{19}$$

For clauses of type I we let $w(C) = \emptyset$. Finally, the work of a set of clauses is the union of the work of each clause in the set.

From this definition it is not difficult to check that, indeed, $w(S_n) = \emptyset$; while the work of the empty clause (which is of type II), and therefore of any refutation of S_n, is $\mathbb{B}^n$. A significant ammount of the proof details are actually spent in showing that (1) non-ground clauses have an empty work, and (2) the work function is invariant with respect to the application of substitutions. In the paper's full version we prove the following two lemmas.

Lemma 4. *Let C be any clause in a refutation of S_n. If C is non-ground, then the work $w(C) = \emptyset$.*

Lemma 5. *Let C be any clause in a refutation of S_n and let σ be a substitution. It then follows that $w(C) = w(C\sigma)$.*

The former lemma will allow us to quickly discard many resolution steps as points where the work could increase in a refutation. The later allows us to more easily analize the effect of the remaining inferences, since the substitution that may be required in order to unify and resolve a pair of clauses will not affect the work measure of each individual clause. Having these, we can now prove the following lemma which is the core of the main result.

Lemma 6. *Let C, C_1, and C_2 be clauses in a refutation of S_n such that C is obtained by resolving another pair of clauses C_1 and C_2 with a unifier σ. It then follows that $\delta = w(C) \setminus w(\{C_1, C_2\})$ has at most one element.*

Proof. If C is either non-ground or of type I, then we know, respectively by Lemma 4 and by definition, that $w(C) = \emptyset$ and the result is trivial. We threfore assume that C is a ground clause of type II.

Suppose that C is the result of resolving two clauses of types I and II.

$$\begin{aligned} C_1\sigma &: [l_1'] \rightarrow l_1 \\ C_2\sigma &: l_1 \wedge l_2 \wedge \cdots \wedge l_m \rightarrow h \\ C &: [l_1'] \wedge l_2 \wedge \cdots \wedge l_m \rightarrow h \end{aligned}$$

Take $\bar{b} \in \delta \subseteq w(C)$, in particular, $\bar{b} \in w^c(l_2) \cap \cdots \cap w^c(l_m) \cap w(h)$. However, since $\bar{b} \notin w(C_2)$ and, by Lemma 5, $\bar{b} \notin w(C_2\sigma)$, it must therefore be the case that $\bar{b} \notin w^c(l_1)$ or, equivalently, $\bar{b} \in w(l_1)$. This actually shows that $\delta \subseteq w(l_1)$. Observe, however, that since l_1 appears as the head of clause a clause of type I, its predicate symbol is either p_j or q_n and, since moreover the predicate is ground, $w(l_1)$ will contain at most a single element (c.f. Definition 6).

Suppose that otherwise C is the result of resolving two clauses of type II.

$$\begin{aligned} C_1\sigma &: l_1' \wedge \cdots \wedge l_{m'}' \rightarrow l_1 \\ C_2\sigma &: l_1 \wedge l_2 \wedge \cdots \wedge l_m \rightarrow h \\ C &: l_1' \wedge \cdots \wedge l_{m'}' \wedge l_2 \wedge \cdots \wedge l_m \rightarrow h \end{aligned}$$

Exactly as in the previous case we can prove for any $\bar{b} \in \delta$ that also $\bar{b} \in w(l_1)$. Now, however, from $\bar{b} \in w(C)$ we also get $\bar{b} \in w^c(l_1') \cap \cdots \cap w^c(l_{m'}')$ and, from this, that $\bar{b} \in w(C_1\sigma) = w(C_1)$, contradicting the hypothesis that $\bar{b} \notin w(C_1)$. The conclusion is that, in this case, $\delta = \emptyset$. □

Our main theorem now follows as a simple corollary of the previous result.

Corollary 2. *A resolution refutation of S_n has a length of, at least, 2^n.*

Proof. Let $\Gamma = C_1, \ldots, C_m$ be a refutation of S_n, i.e. $C_m = \bot$. Now, for every $1 < i \leq m$ let $\delta_i = w(C_1, \ldots, C_i) \setminus w(C_1, \ldots, C_{i-1})$ be the work increment in the proof that is provided by the clause C_i. If C_i is an hypothesis in S_n, then by definition its work is empty and, therefore, $\delta_i = \emptyset$. Suppose that, otherwise, the clause C_i is the result of applying resolution among another pair of clauses C_j and C_k with $i > j$ and $i > k$; as a consequence of Lemma 6 we know that δ_i contains, at most, one element.

Since all elements in $w(\Gamma) = \mathbb{B}^n$ had to be incorporated one element at a time, it therefore follows that the length of the proof, m, is at least 2^n. □

4 Related Work

Although quite recent, there has been a significant interest of the automated reasoning community on effectively propositional logic. The CADE ATP System Competition, since its JC instalment in 2001, holds a division for EPR problems [13]. Moreover, a number of theorem provers particularly geared towards this class of formulae have also been developed. These include, for example, EGROUND by Schulz [17], PARADOX by Claessen and Sörensson [3], DARWIN by Fuchs [5], and IPROVER by Korovin [8].

Systems such as EGROUND and PARADOX implement an instantiation approach where ground instances of the given input formula are generated and then tests for satisfiability are run by a SAT solver. Several ideas have been then proposed in order to limit the number of instances that have to be generated. The work of Schulz [17] discusses and compares some of the early approaches, while Claessen and Sörensson [3] introduce the notion of *sort inference* in the context of *MACE-style* model finding..

In a closely related line of research, the notion of hyper-linking [9, 14] was also proposed in order to restrict the kind of inferences that need to be performed in instantiation based methods. Other related ideas can be found in the work of Hooker et al. [7]. Alternatively, Ganzinger and Korovin [6] propose the use of first-order reasoning (e.g. resolution) to drive the instantiation process; while Baumgartner and Tinelli [1] try to avoid direct instantiation by lifting the propositional DLL algorithm [4] to the first-order level.

Voronkov [21] proposed to use relational databade technology and equality constraints for implementing resolution on EPR formulae and shows that set-at-a-time resolution related operations, such as resolution and subsumption, can be implemented using database operations, for example, joins. A similar observation was later made by Tammet and Kadarpik [20].

Combinations of tableaux related techniques with propositional satisfiability checking have also been proposed by researchers such as Billon [2], Letz and Stenz [10].

5 Conclusion and Future Work

We proved several results showing potential of non-ground methods for proving EPR formulae. First, we proved that resolution can have exponentially shorter proofs than propositional procedures for EPR. Second, we proposed a generalisation rule and showed that resolution with generalisation can have exponentially shorter proofs than resolution.

We find the generalisation rule especially appealing for practical theorem proving. Implementation techniques and heuristics for applying this rule should be developed. Further, it is interesting to investigate soundness of stronger versions of generalisation, both theoretically and practically. In practice, one can try to use potentially unsound versions of this rule and then see if a refutation obtained by a potentially unsound version can be made into a valid refutation.

A crucial step in the efficiency of the generalisation rule is to find new sort inference techniques that restrict sorts as much as possible yet are still sound. We show how some existing sort inference techniques, such as those developed for grounding-based approaches, can be directly applied in this context; while some others, particularly based on linking restrictions, cannot be used in a sound way as easily.

Incidentally, the proofs of these two exponential gaps provide us with benchmark families that might be interesting to test with existing systems. We have shown that, when reasoning with a particular inference system, some unsatisfiable problems have rather short refutations, but are existing implementations of theorem provers able to find such short proofs? Or, which heuristics can we use in order to find these shorter proofs with higher probability?

Further directions for future work include the research on techniques for efficiently implementing and integrating the proposed generalisation inference rule with other systems which already make use of resolution, such as iProver [8] or Vampire [16]. Alternatively, it might also prove fruitful to investigate possible extensions of this inference rule in order to make use of complete linking information which can perhaps better describe the underlying structure of the problem being solved.

References

[1] Baumgartner, P., Tinelli, C.: The model evolution calculus. In: Baader, F. (ed.) CADE 2003. LNCS (LNAI), vol. 2741, pp. 350–364. Springer, Heidelberg (2003)

[2] Billon, J.-P.: The disconnection method. A confluent integration of unification in the analytic framework. In: TABLEAUX 2006: Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods, Terrasini, Italy. LNCS (LNAI), vol. 1071, pp. 110–126. Springer, Heidelberg (1996)

[3] Claessen, K., Sörensson, N.: New techniques that improve MACE-style model finding. In: MODEL 2003: Proceedings of the Workshop on Model Computation (2003)

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

[5] Fuchs, A.: Darwin: A theorem prover for the model evolution calculus. Master's thesis, University of Koblenz-Landau (2004)

[6] Ganzinger, H., Korovin, K.: New directions in instantiation-based theorem proving. In: LICS 2003: Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science, Ottawa, Canada, June 2003, vol. 2, p. 55. IEEE Computer Society, Los Alamitos (1884)

[7] Hooker, J.N., Rago, G., Chandru, V., Shrivastava, A.: Partial instantiation methods for inference in first order logic. Journal of Automated Reasoning 28(3), 371–396 (2002)

[8] Korovin, K.: Implementing an instantiation-based theorem prover for first-order logic. In: Benzmueller, C., Fischer, B., Sutcliffe, G. (eds.) IWIL 2006: Proceedings of the 6th International Workshop on the Implementation of Logics, held at the 13th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2006). CEUR Workshop Proceedings, Phnom Penh, Cambodia, vol. 212 (2006), http://www.cs.man.ac.uk/~korovink/iprover/
[9] Lee, S.-J., Plaisted, D.A.: Eliminating duplication with the hyper-linking strategy. Journal of Automated Reasoning 9(1), 25–42 (1992)
[10] Letz, R., Stenz, G.: DCTP: A disconnection calculus theorem prover. In: Goré, R.P., Leitsch, A., Nipkow, T. (eds.) IJCAR 2001. LNCS (LNAI), vol. 2083, pp. 381–385. Springer, Heidelberg (2001)
[11] Navarro Pérez, J.A., Voronkov, A.: Encodings of bounded LTL model checking in effectively propositional logic. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 346–361. Springer, Heidelberg (2007)
[12] Pérez, J.A.N., Voronkov, A.: Planning with effectively propositional logic. In: Collection of Papers Dedicated to Harald Ganzinger's Memory (to appear, 2007)
[13] Pelletier, F.J., Sutcliffe, G., Suttner, C.B.: The development of CASC. AI Communications 15(2), 79–90 (2002)
[14] Plaisted, D.A., Zhu, Y.: Ordered semantic hyper-linking. Journal of Automated Reasoning 25(3), 167–217 (2000)
[15] Plaisted, D.A., Zhu, Y.: The Efficiency of Theorem Proving Strategies. Computational Intelligence. Vieweg, Wiesbaden, Germany (1997)
[16] Riazanov, A., Voronkov, A.: The design and implementation of Vampire. AI Communications 15(2), 91–110 (2002)
[17] Schulz, S.: A comparison of different techniques for grounding near-propositional CNF formulae. In: Haller, S., Simmons, G. (eds.) FLAIRS 2002: Proceedings of the Fifteenth International Florida Artificial Intelligence Research Society Conference, pp. 72–76. AAAI Press, Menlo Park (2002)
[18] Sutcliffe, G.: The CADE-21 ATP system competition website (2007), http://www.cs.miami.edu/~tptp/CASC/21/
[19] Sutcliffe, G., Suttner, C.B.: The TPTP problem library: CNF release v1.2.1. Journal of Automated Reasoning 21(2), 177–203 (1998)
[20] Tammet, T., Kadarpik, V.: Combining an inference engine with database: A rule server. In: Schröder, M., Wagner, G. (eds.) RuleML 2003. LNCS, vol. 2876, pp. 136–149. Springer, Heidelberg (2003)
[21] Voronkov, A.: Merging relational database technology with the constraint technology. In: Perspectives of System Informatics. Andrei Ershov Second International Memorial Conference (Preliminary Proceedings), Novosibirsk, Akademgorodok, Russia, June 1996, pp. 189–195 (1996)

MaLARea SG1 - Machine Learner for Automated Reasoning with Semantic Guidance

Josef Urban[1,⋆], Geoff Sutcliffe[2], Petr Pudlák[1], and Jiří Vyskočil[1]

[1] Charles University, Czech Republic
[2] University of Miami, USA

Abstract. This paper describes a system combining model-based and learning-based methods for automated reasoning in large theories, i.e. on a large number of problems that use many axioms, lemmas, theorems, definitions, and symbols, in a consistent fashion. The implementation is based on the existing MaLARea system, which cycles between theorem proving attempts and learning axiom relevance from successes. This system is extended by taking into account semantic relevance of axioms, in a way similar to that of the SRASS system. The resulting combined system significantly outperforms both MaLARea and SRASS on the MPTP Challenge large theory benchmark, in terms of both the number of problems solved and the time taken to find solutions. The design, implementation, and experimental testing of the system are described here.

1 Introduction

In recent years the ability of systems to reason over large theories – theories in which there are many functors and predicates, many axioms of which typically only a few are required for the proof of a theorem, and many theorems to be proved from the same set of axioms – has become more important. Large theory problems are becoming more prevalent as large knowledge bases, e.g., ontologies and large mathematical knowledge bases, are translated into forms suitable for automated reasoning [12,13,22], and mechanical generation of automated reasoning problems becomes more common, e.g., [4,8]. Over the years there have been regular investigations into techniques for the a priori selection of the necessary axioms, e.g., [9,17,7,21,11], and the use of externally provided lemmas, e.g., [24,5,26,21,10]. Performances on large theory problems in the TPTP library [19] and the associated CADE ATP System Competition [16] show that automated reasoning systems have improved their ability to select and use the necessary axioms.

The work described in this paper addresses the issue of selecting necessary axioms, from a large set also containing superfluous axioms, to obtain a proof of a conjecture. It combines ordering of axioms by machine learning their relevance to features of conjectures, as found in the MaLARea system [23], with use of

⋆ Supported by a Marie Curie International Fellowship within the 6th European Community Framework Programme.

semantic guidance to advise on the selection of axioms, as found in the SRASS system [18]. The resulting system significantly improves the performance of the two above mentioned systems on the MPTP Challenge[1] large theory benchmark, both in terms of the number of problems solved, and in the speed of processing.

This paper is organized as follows: Section 2 describes the machine learning approach. Section 3 describes the semantic selection technique. Section 4 explains how these two have been combined. Section 5 provides a detailed example of how the combined system works. Section 6 mentions some other improvements that are relevant to the performance of the system. Section 7 gives test results, with commentary. Section 8 concludes and discusses possible future research.

2 Machine Learning Axiom Relevance

The basic idea of the machine learning approach is to interleave ATP system runs on many related problems with learning from successful proofs, and to use the learned knowledge for limiting the set of axioms given to the ATP systems in the following runs. In general, the goal is to learn a mapping from features (in machine learning terminology) of the conjectures (or even of the whole problems when speaking generally) to proving methods that are successful when the features are present. This general setting was instantiated in the first MaLARea version in the following way: The features characterizing conjectures are the symbols appearing in them, and the proving method is ordering of axioms according to their expected relevance to proving the conjecture.

The SNoW machine learning system [2] is used by MaLARea in naive Bayes mode, producing an updated Bayes net after each run, linking symbols with the axioms that were useful for proving conjectures containing those symbols. For learning and evaluation in SNoW, all symbols and axiom names are disjointly translated to integers. The integers corresponding to symbols are the input features, and those corresponding to axioms are the output features. As ATP backends the E prover [15] and SPASS [25] are used by default.

MaLARea is highly configurable, but the default algorithm is as follows: Minimal and maximal number of axioms and time limits are given (4 and 256 axioms, and 1 and 64 seconds by default) – these ranges have been established through experience with the MPTP Challenge problems. First, all ATP backends (E and SPASS by default) are run on all problems (an ATP *problem* is the conjecture and a selection of axioms), using all the axioms and the maximal time limit, to produce as much base knowledge as possible. Second, SNoW is trained on examples saying that each axiom is useful for proving itself, i.e., an example created from an axiom A contains the symbols appearing in A as input features, and A itself as the output feature. This Bayes net is evaluated on each conjecture's symbols, producing an ordering of axioms for each conjecture, according to their symbol overlap with the conjecture. For each conjecture its axiom ordering is used to select the maximal number of axioms, and the resultant problem is given to the ATP systems one-at-a-time, with the minimal time limit. If any

[1] http://www.tptp.org/MPTPChallenge/

ATP system returns a countersatisfiable result[2], then no further proof attempts are made on the problem. Successful proofs are turned into training examples containing the conjectures' symbols as input features, and the axioms used in the corresponding proofs as output features. These training examples are added to the previous examples, and a new Bayes net is trained from them all. The evaluation of the Bayes net on unproved conjectures, selection of axioms, ATP proof attempts, and retraining, iterates. The number of axioms selected and time limit used is adapted according to whether or not any proofs are found in each iteration. When a proof is found in a pass, the axiom and time limits drop to their minimal values for the next pass (hoping that the relevance update caused by the new solution will make some more problems easy to solve). If no proofs are found in a pass, first the axiom limit is doubled (hoping that considering more axioms will help) until the maximal axiom limit is reached. After that the the time limit is quadrupled, hoping that "harder thinking" will help, until the maximal time limit is reached. The algorithm stops when the maximal axiom and time limits are reached.

One (default) change to this basic algorithm is a countersatisfiability precheck, to avoid axiom set selections that do not make sense in the light of previous results. If a new selection of axioms for a countersatisfiable problem's conjecture is a subset of the countersatisfiable problem's axioms, it makes no sense to form a new problem with the new selection of axioms. In this situation, if the time limit is sufficiently low (1s by default), axioms are added in order of their relevance until the new set is no longer a subset of any set which produced a countersatisfiable result for the conjecture. In such cases the axiom limit is "soft", and may be violated.

3 Semantic Relevance Axiom Selection

The semantic relevance selection algorithm implemented in SRASS selects axioms for each conjecture independently, with no benefit accrued from previous successful selections. For each problem it starts with an empty set of selected axioms. At each pass the process looks for a model of the selected axioms and the negation of the conjecture. If no such model exists then the conjecture is a logical consequence of the selected axioms. If such a model exists then an unselected axiom that is false in the model is moved to the set of selected axioms. The newly selected axiom excludes the model (and possibly other models) from the set of models of the selected axioms and negated conjecture, eventually leading to the situation where there are no models of the selected axioms and the negated conjecture. The Figure 1 shows the idea. The plane represents the space of interpretations, the rectangle encompasses the models of the conjecture C, and an oval encompasses the models of the corresponding axiom A_i. In the first

[2] Status values, such as countersatisfiable, are taken from the SZS ontology [20]. The ontology provides result and output forms that specify what has been established about an ATP problem. For example, countersatisfiable problems have a non-provable conjecture.

pass, when the set of selected axioms is empty, the model M_0 of the negation of the conjecture, $\neg C$, is found. That leads to the selection of the axiom A_1, which is false in the model. Iteratively, the model M_1 of $\{A_1, \neg C\}$ is found, leading to the selection of A_2, the model M_2 of $\{A_1, A_2, \neg C\}$ is found, leading to the selection of A_3, at which point there is no model of $\{A_1, A_2, A_3, \neg C\}$, proving that C is a logical consequence of $\{A_1, A_2, A_3\}$. In the last part of the figure this is seen by the intersection of the axiom ovals lying within the conjecture rectangle.

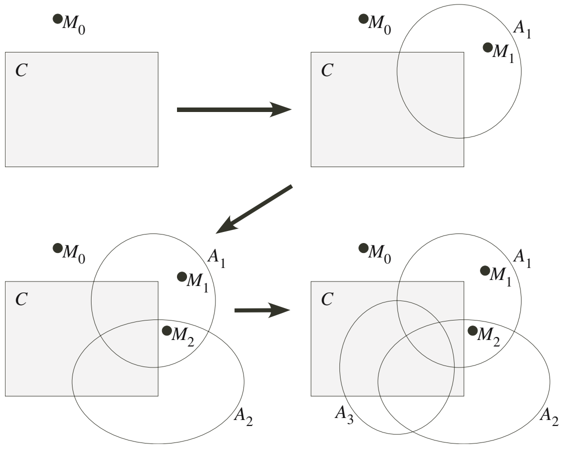


Fig. 1. The Basic Process

Example: Consider the simple propositional problem, to prove the conjecture $C = b$ from the axioms $E_1 = a \mid b$, $E_2 = b \Rightarrow a$, $E_3 = (\neg a \; \& \; (b \mid c)) \mid (a \; \& \; \neg b \; \& \; \neg c)$. and $E_4 = b \mid (a \Leftrightarrow c)$. The conjecture C can be proved from E_3 and E_4, i.e., E_1 and E_2 are superfluous. The following table shows a possible sequence of models and selected axioms. Note how E_1, E_2, and E_3 are true in the first model, so only E_4 can be selected. E_1 and E_3 are false in the second model, but E_1 is found first. E_2 is true in every model of $\neg C$, and thus can never be selected. If the model $\{a, \neg b, c\}$ had been used in the second pass, then E_3 would have been selected, leading to immediate success.

	Selected set	Model	Axiom
1	$\{\ \}$	$\{a, \neg b, \neg c\}$	$E_4 = b \mid (a \Leftrightarrow c)$
2	$\{E_4\}$	$\{\neg a, \neg b, \neg c\}$	$E_1 = a \mid b$
3	$\{E_1, E_4\}$	$\{a, \neg b, c\}$	$E_3 = (\neg a \; \& \; (b \mid c)) \mid (a \; \& \; \neg b \; \& \; \neg c)$
4	$\{E_1, E_3, E_4\}$	-	-

The order in which the axioms are considered at each pass of this process is computed in at the start, according to the *contextual indirect relevance* of each axiom to the conjecture. for the axiom to contribute to a proof of a conjecture F_c, in the context of a set S of axioms and the conjecture. First, the contextual direct relevance between all formulae pairs $F_1, F_2 \in Axioms \cup \{Conjecture\}$ is measured by

$$\frac{\sum_{s \in (sym(F_1) \cap sym(F_2))} \left(1 - \frac{|\{f : f \in S, s \in sym(f)\}|}{|S|}\right)}{|sym(F_1) \cup sym(F_2)|}$$

where $sym(F)$ is the set of function and predicate symbols occuring in F. Next, the contextual path relevance of every path $F_a = F_1 \cdot F_2 \cdot \ldots \cdot F_n = F_c$ from an axiom F_a to the conjecture F_c is calculated as the smallest contextual direct relevance in the path, divided by the length of the path. Finally, the contextual indirect relevance between F_a and F_c is taken as the maximal contextual path relevance over all paths connecting F_a to F_c.

In the SRASS implementation a range of further extensions are used to improve performance, but these are not relevant here.

4 Combining Machine Learning with Semantic Selection

The addition of semantic selection to MaLARea consists of three extensions to the selection of axioms based on relevance. To implement the extensions a good model finder, Paradox [3], is added to the set of ATP backends.

The first extension is very simple: it checks for countersatisfiability in runs where this is probable – currently when the time limit is 1s and at most 64 axioms are selected. Paradox can detect countersatisfiability much more effectively than E and SPASS, especially if there are very few axioms, and thus eliminates more hopeless proof attempts. This in turn allows the countersatisfiability precheck to detect more cases when more axioms need to be added.

The second extension uses the models found when a problem is found to be countersatisfiable, as an additional criterion for computing axiom relevance. For this extension it is necessary to efficiently evaluate formulae in the models. This (a bit surprisingly) turns out to be a problem both with the Paradox and Darwin [1] systems. The only existing tool for this task that we know of is Bill McCune's `clausefilter` program, which comes with the Mace4 [6] model finder. Unfortunately, the model formats used by Paradox and Mace4 are incompatible. Paradox is more successful than Mace4 (one of the reasons is that Mace4 avoids models of cardinality 1), and so more useful for the first extension. Therefore, Mace4 is run after Paradox, and only when Paradox finds a model. If Mace4 finds a model, the model is stored, and the truth values of all axioms and conjectures are computed wrt the model by `clausefilter`. This is very fast, and poses no efficiency problems. Often some of the axioms and conjectures cannot be evaluated in the model, because they contain symbols not interpreted by the model. The values of such formulae are therefore undefined wrt the model. For each model, the set of symbols it interprets, the set of true formulae, and the set of false formulae, are recorded. The models are serially numbered, and a hash value (just a SHA1 (160-bit) checksum) is recorded to quickly detect duplicate models. For each formula a vector of its models and a vector of its countermodels (models in which it is false) are recorded.

The stored models are used following the axiom selection approach described in Section 3, i.e., excluding models of the axioms and negated conjecture by adding axioms that are false in the models. Given a conjecture C, if an axiom is false in a model of $\neg C$ then the axiom excludes that model. The more such models an axiom excludes, the more it is likely to be useful. Also, if only a few axioms exclude a model, those axioms are more likely to be useful. This kind of model-based relevance is implemented in the same way as the symbol-based relevance. For each axiom and conjecture a training example is produced. The example has the countermodels (again encoded as integers) of the formula as input features, and the formula itself as the output feature. These examples are added to those described in Section 2. The resulting Bayes net is then evaluated on the features of unproved conjectures, i.e., the symbols and countermodels of the conjecture. This leads to increased relevance of axioms that are false in many (and more rare) countermodels of the conjecture. The relevance boost caused by the model information can be quite big, see, e.g., the promotion of axiom `t143_relat_1` in Section 5.

The third extension goes one step closer to the model-based algorithm described in Section 3. It complements the second extension in way that is analogous to the way that the subset precheck complements the machine learned selection of axioms – by extending the axiom specification using a "logical" criterion. This criterion is as follows: the set of axioms should exclude as many known models of the negated conjecture as possible. While the second extension only promoted the relevance of axioms that contribute to this goal, this extension attempts to meet this goal as much as possible. Given a conjecture C and an axiom selection A, the set NM of the models of $\neg C$ is retrieved. Models already excluded by axioms in A (countermodels of axioms in A) are deleted from NM. If some models of $\neg C$ remain after this, the remaining axioms are inspected in their relevance order, and included into the specification if they exclude additional models of $\neg C$. This continues until either all models of $\neg C$ are excluded by the axioms, or there are no more axioms available. The complete algorithm, combining the axiom relevance and semantic selection, is shown in Figure 2. See Section 5 for a detailed example of how this algorithm proceeds, and particularly Section 5.1 for discussion of some choices in its implementation.

The semantic selection in MaLARea has several significant differences to that of SRASS. While syntactic relevance ordering of axioms is used there in the beginning, it remains static. Here the relevance ordering is continuously updated, taking into account successful proofs and model-based relevance. The models created in all proof attempts here are immediately used for evaluation of all axioms and conjectures, and this information is re-used many times in many problems. Multiple axioms can be selected to extend the specifications by the model reasoner in one pass, because the whole database of models is taken into account. An important byproduct of the algorithm is the evolving database of interesting and relevant models. Its final size for the experiments conducted in Section 7 is ca. 1000 models. The models are interesting and relevant because they are products of proof attempts based on the "world knowledge" accumulated in previous passes. To a large extent the model reasoner prevents creation of the same model twice in two different passes: new models typically have to have a new characteristic vector of truth values on all formulae. One possible future use of the model database is for semantic detection of interesting lemmas and conjectures: a lemma or conjecture is interesting when its characteristic vector on the database of models differs from others.

5 Detailed Example

The methods described in Section 4 are illustrated on the proof of the theorem `t144_relat_1`, during the system's processing of the MPTP Challenge problems. The theorem says that an image of a set under a relation is a subset of the relation's range. In TPTP and Mizar notations it is as follows:

```
! [A,B] : (relation(B) => subset(relation_image(B,A), relation_rng(B)))
for X being set for R being Relation holds R .: X c= rng R
```

The axioms named below are available in TPTP form from the MPTP Challenge distribution, and in Mizar form from Mizar proofs of the MPTP Challenge problems.[3] This theorem was proved in the fifth 1 second pass of the system. In the first pass all 198 axioms were considered, while in the remaining four passes the limit of four axioms was used. In the end the theorem was proved from the axioms `d3_tarski, t143_relat_1, t20_relat_1`.

The axioms used in the second to fifth passes were:

```
2: t25_relat_1,t99_relat_1,t118_relat_1,t45_relat_1
3: t99_relat_1,t88_relat_1,t20_relat_1,t45_relat_1,t116_relat_1,t44_relat_1
4: t20_relat_1,t99_relat_1,t88_relat_1,t45_relat_1,d3_tarski,t116_relat_1,
   t44_relat_1
5: t20_relat_1,d3_tarski,t99_relat_1,t143_relat_1,t45_relat_1,
   reflexivity_r1_tarski
```

In pass 2 the soft limit of four axioms was observed, while in the other three the limit was violated – two or three extra axioms were added by the subset precheck and by model reasoning to the four axioms selected by relevance. In pass 3 the four axioms `t99_relat_1, t88_relat_1, t20_relat_1, t45_relat_1` were selected initially, by their relevance from previous successful proofs and by their symbolic and structural similarity with the conjecture. The resultant problem was not known to be countersatisfiable, so there was no need to add axioms for that reason. However in pass 2 the following four models of the negated conjecture `t144_relat_1` had been found (we only give the model numbers, conjecture that lead to their discovery, and formulae that were evaluated as false in them by `clausefilter`):

[3] `http://lipa.ms.mff.cuni.cz/~urban/xmlmml/html_abstr.930/00mptp_chall_topdown.html`

```
Input: $MinAxioms$, $MaxAxioms$, $MinCPU$, $MaxCPU$, $CSACPULimit$,
$Conjectures$, $AllAxioms$, for each $C \in Conjectures$ $Axioms(C) \subseteq AllAxioms$
$TrainingSet = \emptyset$, $NumAxioms = MaxAxioms$, $CPULimit = MinCPU$;
forall $F \in (AllAxioms \cup Conjectures)$ do
    $CMOSet(F) = \emptyset$ ;                    // The CounterModels of $F$
    $TrainingSet = TrainingSet \cup \{\langle Symbols(F), F\rangle\}$;
end
forall $C \in Conjectures$ do
    $CSASet(C) = \emptyset$ ;     // The CounterSatisfiable axiom sets for $C$
    $Result$ = Run systems on $Axioms(C) \models C$ with $MaxCPU$;
    $ProcessResult(C, Result, Axioms(C))$;
end
Bayes net $B$ = naive Bayes learning on $TrainingSet$;
while $NumAxioms \leq MaxAxioms$ or $CPULimit \leq MaxCPU$ do
    forall unproved $C \in Conjectures$ do
        Evaluate $B$ on $Symbols(C) \cup CMOSet(C)$ to order $Axioms(C)$;
        $UseAxioms$ = $NumAxioms$ most relevant axioms from $Axioms(C)$;
        if $CPULimit < CSACPULimit$ then
            while $\exists S \in CSASet(C)$: $UseAxioms \subseteq S$ do
                Add next most relevant axiom to $UseAxioms$;
            end
        end
        while $\exists M \in CMOSet(C)$: $\neg(\exists A \in UseAxioms$: $M \models \neg A)$ do
            Add next most relevant $A$: $M \models \neg A$, to $UseAxioms$;
        end
        $Result$ = Run systems on $UseAxioms \models C$ with $CPULimit$;
        $ProcessResult(C, Result, UseAxioms)$;
    end
    Bayes net $B$ = naive Bayes learning on $TrainingSet$;
    if Any result is Theorem then
        $NumAxioms = MinAxioms$, $CPULimit = MinCPU$;
    else if $NumAxioms < MaxAxioms$ then
        $NumAxioms = 2 * NumAxioms$;
    else $CPULimit = 4 * CPULimit$;
end
Procedure $ProcessResult(C, Result, AxiomsTried)$:
if $Status(Result)$ = Theorem then
    $TrainingSet = TrainingSet \cup \{\langle Symbols(C), AxiomsInProof(Result)\rangle\}$;
else if $Status(Result)$ = CounterSatisfiable then
    $CSASet(C) = CSASet(C) \cup \{AxiomsTried\}$;
    forall $M \in ModelsFound(Result)$ do
        forall $F \in (AllAxioms \cup Conjectures)$: $M \models \neg F$ do
            $CMOSet(F) = CMOSet(F) \cup \{M\}$;
            $TrainingSet = TrainingSet \cup \{\langle M, F\rangle\}$;
        end
    end
end
```

Fig. 2. MaLARea Algorithm

```
- 22; t144_relat_1; reflexivity_r1_tarski,t116_relat_1,t144_relat_1
- 23; t145_funct_1; d10_xboole_0,d1_relat_1,d3_relat_1,d3_tarski,
  reflexivity_r1_tarski,t144_relat_1,t145_funct_1,t167_relat_1,t2_xboole_1,
  t33_zfmisc_1,t88_relat_1,t99_relat_1
- 26; t146_relat_1; d10_xboole_0,reflexivity_r1_tarski,t144_relat_1,
  t146_relat_1,t160_relat_1,t2_xboole_1,t44_relat_1,t45_relat_1
- 32; t160_relat_1; d10_xboole_0,dt_k5_relat_1,reflexivity_r1_tarski,
  t144_relat_1,t160_relat_1,t2_xboole_1,t3_xboole_1,t44_relat_1
```

The model reasoner looked at the four suggested axioms, and saw that models 22 and 32 were not excluded by them, i.e., none of the four axioms were false in those two models of the negated conjecture, while model 23 was already excluded by axiom `t88_relat_1`, and model 26 by axiom `t45_relat_1`. The model reasoner therefore started to go through additional axioms in their order of relevance, and found that axiom `t116_relat_1` excluded model 22, and axiom `t44_relat_1` excluded model 32. At this point all known models of the negated conjecture were excluded, so this axiom selection was used for pass 3. The resultant problem was found to be countersatisfiable, yielding the following model (and this was the only new model of the negated conjecture `t144_relat_1` found in pass 3):

```
- 181; t144_relat_1; d3_tarski,reflexivity_r1_tarski,t144_relat_1
```

The relevance based selection suggested these four axioms for pass 4:

```
  t20_relat_1,t99_relat_1,t88_relat_1,t45_relat_1
```

which is just a permutation (caused by relevance update) of the four initial axioms suggested in pass 3. Since the previous problem (containing two additional axioms) was known (now) to be countersatisfiable, the subset precheck caused the addition of axiom `d3_tarski`, at which point the problem was not known to be countersatisfiable. There were two reasons for this axiom becoming very relevant at this point, and being the next in the relevance queue: The first reason was the "standard learning" argument. In pass 3, this axiom (definition of set-theoretical inclusion) was successfully used to prove several theorems, raising its overall relevance. The second reason was that this axiom became known to be false in a new model (181) of the negated conjecture, and this fact was taken into account by the relevance learner. After this addition by the the subset precheck, the model reasoner inspected the axiom selection. Since only one new model (181) was added in pass 3, and that model was already excluded by `d3_tarski` (which did not exclude any of the old models), it just proceeded to exclude the same two remaining models (22 and 32) by adding the same two axioms `t116_relat_1` and `t44_relat_1` axioms as before. In pass 4 this problem was found to be countersatisfiable, and three more models of the negated conjecture were produced:

```
- 315; t144_relat_1; t143_relat_1,t144_relat_1
- 316; t145_funct_1; d12_funct_1,d13_relat_1,d1_relat_1,t143_relat_1,
  t144_relat_1,t145_funct_1,t146_relat_1,t147_funct_1,t33_zfmisc_1
- 324; t160_relat_1; d10_xboole_0,d13_relat_1,d1_relat_1,d3_relat_1,
  d3_tarski,d4_relat_1,d5_relat_1,d8_relat_1,dt_k5_relat_1,
  reflexivity_r1_tarski,t144_relat_1,t160_relat_1,t2_tarski,t33_zfmisc_1
```

The relevance based selection suggested these four axioms for pass 5:

```
t20_relat_1,d3_tarski,t99_relat_1,t143_relat_1
```

Note that `d3_tarski` and `t143_relat_1` replaced `t88_relat_1` and `t45_relat_1` in the top four positions of the relevance order. In case of `d3_tarski` this is again both because it was successfully used to prove some more other theorems in pass 4, and because it now excludes one more model (324) of the negated conjecture. In case of `t143_relat_1` it is solely because it excludes two fresh models (315, 316) of the negated conjecture (and especially the exclusion of the model 315 is a relatively strong relevance boost, because there are only two formulae with that feature, and this rarity is taken into account by the Bayes learner). This basic specification was not known to be countersatisfiable, so the model reasoner started to inspect it. The four initial axioms already excluded models 23, 181, 315, 316 and 324. Models 22, 26, and 32 were not excluded by any of these four axioms, so first the axiom `t45_relat_1` was added, excluding model 26, and then the axiom `reflexivity_r1_tarski` was (finally) sufficiently relevant and excluded the remaining models 22 and 32. This problem produced the proof, using axioms `d3_tarski`, `t143_relat_1`, and `t20_relat_1`.

It is interesting to note in the pass 3 that the axiom `reflexivity_r1_tarski` excluded all of the four models, but it was not sufficiently relevant to be selected. Similarly, in pass 4, `reflexivity_r1_tarski` excludes all of the models. This suggests that an alternative model covering algorithm could be useful for making the specifications smaller. Preliminary experiments suggest that using, e.g., MiniSat+, for finding the optimal model coverings would be feasible. It might also be worth experimenting with exchanging relevance-greed for covering-greed, i.e., taking the axioms that exclude most candidate models first.

5.1 Partial vs. Total Models

The reader might wonder why the model reasoner added the axioms `t45_relat_1` and `reflexivity_r1_tarski` (this one alone would be sufficient, as it also excludes model 26) in the last pass. These axioms were not needed for the final proof, i.e., in every model of the negated conjecture at least one of the necessary axioms (`d3_tarski`, `t143_relat_1`, and `t20_relat_1`) should be false. Why then don't they exclude models 22, 26, and 32? The answer is in the implementation of models. The models produced by Mace4 are limited to the interpretation of symbols that appear in the problem formulae. The axiom `d3_tarski` defines `subset` in terms of `in` (membership), but `in` did not appear in the problems that led to the construction of models 22, 26, and 32. Similarly for `t143_relat_1`. Therefore the truth values of these two axioms are undefined in the three models.

This points to another interesting choice: For efficient proving and model finding we want to constrain the relevant set of symbols as much as possible. Often this leads to success – the number of theorems that can be proved by exploring Boolean properties of set-theoretical union, inclusion, difference, and subset, without any word about membership, is a good example. On the other hand, the notion of "relevance" evolves as theorems are proved, and it may be necessary expand some of the definitions, as happened in the proof of `t144_relat_1`.

Such expansion, however, makes some of the earlier model-based information inaccurate in some sense.

There are several ways to cope with these problems. The current solution is just to ignore it: the worst that can happen is that extra axioms are added by the model reasoning. One (quite feasible) alternative is to try to extend all models to the set of "relevant symbols", defined as the union of symbols appearing in the axioms that are above some relevance threshold, or the union of symbols appearing in all relevant models, etc. The model extension could be done by adding tautologies containing the new symbols, or by manually adding some default interpretation of the new symbols to the existing models, providing a default way for extending partial models to total models. The danger of this approach is that such total models are mostly interesting on only their partial symbol domain, and evaluation of many formulae outside that domain will flood the system with a lot of boring values. Another interesting approach could be to watch the relevance of models in a way similar to watching the relevance of axioms, and preferring evaluation in more relevant models. This option is likely to be used anyway, if the number of generated models gets too high, and evaluation of all formulae in them too expensive.

6 Other Performance Related Additions to MaLARea

6.1 Term Structure

It is not difficult to come up with examples when just symbol-based similarity can be too crude. One way to picture this is to realize that all the symbols used in the MPTP Challenge problems are introduced by their set-theoretical definitions that can recursively be expanded to just the basic set theory level, i.e., to formulae containing only the $\in$ and $=$ symbols. If that were done, symbol-based similarity would become completely useless, but the expanded formulae would still be more or less close to one another when considering the structure of terms and subformulae. This suggests that structural similarity is likely to be a useful complement to symbol-based similarity. This is confirmed by the experimental results shown in Section 7. The practical implementation in MaLARea takes an advantage of the shared term mechanisms used in the E prover. E prover keeps only one copy of each term in its shared term bank, and it also treats all formulae for clausification purposes internally as shared terms. The shared terms have internal serial numbering, used, e.g., for compact printing. Given these nice functionalities, it was straightforward to write a small E-based program that reads all formulae appearing in all problems into a shared term bank, and then for each formula it prints the internal numbers of all the shared terms (i.e., also subformulae) appearing in it. The information is printed in the following form:

```
terms(t144_relat_1,[492,491,8325,8324,1803,2153]).
```

There are about 10000 shared terms in the MPTP Challenge formulae (compared to 418 symbols), giving a much finer similarity measure that can be used for learning in the same way as the symbol features. One additional modification was implemented to accommodate the problem of different (normalized) variables appearing in the shared terms, making terms that differ only by variable renaming have different numbers. This was solved by renaming all variables in each formula to just one variable before handing the formulae to the shared term extractor. MaLARea has options for specifying all these similarity measures (symbols, variable-renamed shared terms, standard shared terms), and their combinations.

6.2 ATP Systems

The Vampire [14] system was previously not usable in MaLARea, because its proof output was hard to parse. Therefore TPTP format output was added to Vampire, allowing its use as an ATP backend along with E and SPASS. First experiments on the MPTP Challenge show that using Vampire does not help the combination of E and SPASS. One reason can be traced back to Vampire's tuning on the TPTP library: the Fampire experiment (using Vampire with a different clausifier) significantly improves Vampire's performance on the MPTP Challenge problems. This is caused by a different (much better for MPTP) set of strategies being used by Vampire for CNF problems than the strategies used by Vampire for FOF problems. This again suggests that addition of ATP strategy learning to MaLARea could be quite useful. Such effort would be greatly facilitated by having a common TPTP format for ATP parameters, and systems like E, SPASS, and Vampire actually using that format.

Other work that had positive impact was the correction of a bug in the proof mode of the E prover. This error had caused the strategies used in the fast "assurance" mode to be replaced by dummy strategies in proof mode, often causing failure in the proof mode. This in turn prevented learning the axiom relevance from some of the theorems proved by E. This correction improved MaLARea's performance on the MPTP Challenge benchmark by 4 problems.

7 Results

The set of problems from the chainy division of the MPTP Challenge was used for evaluation. This set of problems was created with the intention of providing a benchmark for large theory reasoning. It consists of 252 related mathematical problems, translated by the MPTP system [22] from the Mizar library. The conjectures of the problems are the Mizar theorems that are recursively needed for the Mizar proof of one half (one of two implications) of the general topological Bolzano-Weierstrass theorem. The whole problem set contains 1234 formulae and 418 symbols. The problems in the chainy division intentionally contain all the "previous knowledge" as axioms. This results in an average problem size of ca. 400 formulae. The challenge has an overall time limit policy, i.e., instead of solving problems one-at-a-time with a fixed time limit, the overall time limit of 21 hours can be used in any way for solving the problems. The experiments on this set of problems were done on 2.33GHz Intel Xeons (4MB cache) with 1GB RAM. These machines seem to be about 4% faster than the machines used for evaluating many other system for the MPTP Challenge in 2007.

The first experiment corresponds to the standard MPTP Challenge, i.e., solving as many problems as possible within the overall time limit. Table 1 shows the results of three different runs: MaLARea in the standard setting presented in Section 2[4], MaLARea with the term structure learning presented in Section 6.1, and finally MaLARea with term structure learning and the semantic guidance presented in Section 4. All three runs used the default minimal and maximal time and axiom limits (1-64s, 4-256 axioms, 1s threshold for the countersatisfiability precheck). All of them were left to run beyond the MPTP Challenge overall time limit of 21 hours, until they stopped themselves by reaching the maximal axiom and time limits.

Table 1. Full run of three versions of MaLARea on the MPTP Challenge

description	total solved	solved in 21 hours	passes in 21 hours	total passes	last pass with success	time to stop (hours)	time to solve last
standard	149	144	114	234	208	83	66
with TS	154	149	148	263	237	55	47
with TS & SG	165	161	121	195	170	52	37

The table shows that the addition of term structure improves the original version by 5 problems (both within the 21 hour time limit and overall), and that the semantic guidance then contributes an additional 12 (within the 21 hour time limit), resp. 11 (without time limit) solved problems. Although the numbers of solved problems differ quite significantly, it is also interesting to note that the version with semantic guidance peaked sooner (37 hours to solve all of the 165 problems) than the other versions. This faster convergence becomes even more apparent when we consider just the number of problems solved in the fast 1s passes following the all-axioms 64s first pass. This first pass obviously gave the same values for all three versions (because it uses all the axioms), solving 100 problems, all within 1s. However the numbers of problems solved in the following 1s passes differ quite considerably – see Table 2. This table shows that the version with semantic guidance solved more problems (56) in the following 1s passes than the version without semantic guidance solved in all following passes (154 problems solved - 100 solved in the first pass = 54). It should also be noted that the addition of Paradox typically makes these low-time limit runs faster than in the version without semantic guidance, because problems with few axioms are often countersatisfiable, and Paradox is much better at detecting that than E and SPASS.

Given this good performance with low time limits, it is natural to ask what happens when the time limit is reduced to 1s in the all-axioms first pass. This has a quite pragmatic motivation: both E and SPASS take the full 64s on the 152 problems not solved in the first pass (252 problems, 100 solved in the first pass),

[4] But with visible improvement caused by correction of the E bug. Even without this correction, standard MaLARea was the best performing system on the MPTP challenge at the time of the July 2007 evaluation.

Table 2. Statistics of the Initial 1s Passes

description	total solved	solved in 1st pass	solved in 1s passes	number of 1s passes
standard	149	100	36	36
with TS	154	100	41	56
with SG,TS	165	100	56	54

Table 3. Statistics of the 1s-only passes of the version using TS and SG

pass	ax limit	solved	cumul	pass	ax limit	solved	cumul	pass	ax limit	solved	cumul
1	0	60	60	12	4	3	120	23	4	0	134
2	256	7	67	13	4	1	121	24	8	1	135
3	256	1	68	14	4	2	123	25	4	0	135
4	4	12	80	15	4	3	126	26	8	0	135
5	4	10	90	16	4	0	126	27	16	0	135
6	4	14	104	17	8	2	128	28	32	4	139
7	4	5	109	18	4	2	130	29	4	0	139
8	4	2	111	19	4	0	130	30	8	0	139
9	4	3	114	20	8	0	130	31	16	0	139
10	4	2	116	21	16	3	133	32	32	0	139
11	4	1	117	22	4	1	134	33	64	2	141

so the pass takes more than 5 hours. The second experiment thus imposed a time limit of 1s in the all-axioms first pass. The results are shown in Table 3. Every pass takes 5-10 minutes (depending on how much Paradox is successful, and if (and how quickly) Mace produces a model). The value of 128 solved problems (more than 50%), which is beyond what any other ATP system solves in 21 hours, is reached in about 2-3 hours. The value of 104 solved problems, which is above the combined forces of the two underlying ATPs run both with 64s time limit, can be reached in six passes, i.e., in ca. 30-60 minutes.

8 Future Work and Conclusion

Human mathematics is very often a large theory. Humans are both good at developing large theories, and at reasoning within them. These two capabilities are certainly connected: it takes a lot of theory development to prove major mathematical results like Fermat's last theorem and the Poincare conjecture. The system presented here tries to reason only in a fixed theory. It would be interesting to add theory development to the system. The first try could be just addition of useful lemmas created during the proof attempts. Another line of research is work on the semantic part. Human mathematicians have a well developed store of interesting examples and counterexamples that they use for testing their theories, suggesting conjectures, and getting their proofs right. The models created so far are simple – finite, often of small cardinality. It would be interesting to try to accommodate infinite domains, by generalizing the current simple model reasoning to perhaps less precise, but more sophisticated evaluation algorithms that are probably used by humans.

A lot of work on the system can be done by improving the existing components. Different learning algorithms (for different parts of the system) can be experimented with, different backends can be used in different settings (see, e.g., the note about strategy learning in Section 6.2), and the usage of various components in the overall loop can certainly be experimented with much more. Another challenge is to accommodate large theories that are not so strictly consistent in symbol and axiom naming as the MPTP problems, e.g., by using even more abstract structural information. While many of these extensions are interesting AI research, it is also good to keep in mind that the current performance of the system already makes it useful, e.g., for proof assistance, and to further work on its practical usability by e.g., the Mizar and Isabelle formalizers.

Acknowledgements. Thanks to Bill McCune for his help with various Mace4 and clausefilter issues, and to Stephan Schulz and Christoph Weidenbach for correcting the E and SPASS bugs that influenced the performance of the system.

References

1. Baumgartner, P., Fuchs, A., Tinelli, C.: Darwin - A Theorem Prover for the Model Evolution Calculus. In: Sutcliffe, G., Schulz, S., Tammet, T. (eds.) Proceedings of the Workshop on Empirically Successful First Order Reasoning, 2nd International Joint Conference on Automated Reasoning (2004)
2. Carlson, A., Cumby, C., Rosen, J., Roth, D.: SNoW User's Guide. Technical Report UIUC-DCS-R-99-210, University of Illinois, Urbana-Champaign (1999)
3. Claessen, K., Sorensson, N.: New Techniques that Improve MACE-style Finite Model Finding. In: Baumgartner, P., Fermueller, C. (eds.) Proceedings of the CADE-19 Workshop: Model Computation - Principles, Algorithms, Applications (2003)
4. Denney, E., Fischer, B., Schumann, J.: Using Automated Theorem Provers to Certify Auto-generated Aerospace Software. In: Basin, D., Rusinowitch, M. (eds.) IJCAR 2004. LNCS (LNAI), vol. 3097, pp. 198–212. Springer, Heidelberg (2004)
5. Fuchs, M.: Controlled Use of Clausal Lemmas in Connection Tableau Calculi. Journal of Symbolic Computation 29(2), 299–341 (2000)
6. McCune, W.W.: Mace4 Reference Manual and Guide. Technical Report ANL/MCS-TM-264, Argonne National Laboratory, Argonne, USA (2003)
7. Meng, J., Paulson, L.: Lightweight Relevance Filtering for Machine-Generated Resolution Problems. In: Sutcliffe, G., Schmidt, R., Schulz, S. (eds.) Proceedings of the FLoC 2006 Workshop on Empirically Successful Computerized Reasoning, 3rd International Joint Conference on Automated Reasoning. CEUR Workshop Proceedings, vol. 192, pp. 53–69 (2006)
8. Meng, J., Paulson, L.: Translating Higher-order Problems to First-order Clauses. Journal of Automated Reasoning 40(1), 35–60 (2008)
9. Plaisted, D.A., Yahya, A.: A Relevance Restriction Strategy for Automated Deduction. Artificial Intelligence 144(1-2), 59–93 (2003)
10. Pudlak, P.: Search for Faster and Shorter Proofs using Machine Generated lemmas. In: Sutcliffe, G., Schmidt, R., Schulz, S. (eds.) Proceedings of the FLoC 2006 Workshop on Empirically Successful Computerized Reasoning, 3rd International Joint Conference on Automated Reasoning. CEUR Workshop Proceedings, vol. 192, pp. 34–52 (2006)
11. Pudlak, P.: Draft Thesis: Verification of Mathematical Proofs (2007)

12. Quaife, A.: Automated Development of Fundamental Mathematical Theories. Kluwer Academic Publishers, Dordrecht (1992)
13. Ramachandran, D., Reagan, P., Goolsbey, K.: First-orderized ResearchCyc: Expressiveness and Efficiency in a Common Sense Knowledge Base. In: Shvaiko, P. (ed.) Proceedings of the Workshop on Contexts and Ontologies: Theory, Practice and Applications (2005)
14. Riazanov, A., Voronkov, A.: The Design and Implementation of Vampire. AI Communications 15(2-3), 91–110 (2002)
15. Schulz, S.: E: A Brainiac Theorem Prover. AI Communications 15(2-3), 111–126 (2002)
16. Sutcliffe, G.: The 3rd IJCAR Automated Theorem Proving Competition. AI Communications 20(2), 117–126 (2007)
17. Sutcliffe, G., Dvorsky, A.: Proving Harder Theorems by Axiom Reduction. In: Russell, I., Haller, S. (eds.) Proceedings of the 16th International FLAIRS Conference, pp. 108–112. AAAI Press, Menlo Park (2003)
18. Sutcliffe, G., Puzis, Y.: SRASS - a Semantic Relevance Axiom Selection System. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 295–310. Springer, Heidelberg (2007)
19. Sutcliffe, G., Suttner, C.B.: The TPTP Problem Library: CNF Release v1.2.1. Journal of Automated Reasoning 21(2), 177–203 (1998)
20. Sutcliffe, G., Zimmer, J., Schulz, S.: TSTP Data-Exchange Formats for Automated Theorem Proving Tools. In: Zhang, W., Sorge, V. (eds.) Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems. Frontiers in Artificial Intelligence and Applications, vol. 112, pp. 201–215. IOS Press, Amsterdam (2004)
21. Urban, J.: MizarMode - An Integrated Proof Assistance Tool for the Mizar Way of Formalizing Mathematics. Journal of Applied Logic 4(4), 414–427 (2006)
22. Urban, J.: MPTP 0.2: Design, Implementation, and Initial Experiments. Journal of Automated Reasoning 37(1-2), 21–43 (2006)
23. Urban, J.: MaLARea: a Metasystem for Automated Reasoning in Large Theories. In: Urban, J., Sutcliffe, G., Schulz, S. (eds.) Proceedings of the CADE-21 Workshop on Empirically Successful Automated Reasoning in Large Theories, pp. 45–58 (2007)
24. Veroff, R.: Using Hints to Increase the Effectiveness of an Automated Reasoning Program: Case Studies. Journal of Automated Reasoning 16(3), 223–239 (1996)
25. Weidenbach, C.: Combining Superposition, Sorts and Splitting. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, pp. 1965–2011. Elsevier Science, Amsterdam (2001)
26. Zhang, Y., Sutcliffe, G.: Lemma Management Techniques for Automated Theorem Proving. In: Konev, B., Schulz, S. (eds.) Proceedings of the 5th International Workshop on the Implementation of Logics, pp. 87–94 (2005)

CASC-J4
The 4th IJCAR ATP System Competition

Geoff Sutcliffe

Department of Computer Science, University of Miami

The CADE ATP System Competition (CASC) is an annual evaluation of fully automatic, first-order Automated Theorem Proving (ATP) systems - the world championship for such systems. In addition to the primary aim of evaluating the relative capabilities of ATP systems, CASC aims to stimulate ATP research in general, to stimulate ATP research towards autonomous systems, to motivate implementation and fixing of systems, to provide an inspiring environment for personal interaction between ATP researchers, and to expose ATP systems within and beyond the ATP community. Fulfillment of these objectives provides stimulus and insight for the development of more powerful ATP systems, leading to increased and more effective usage. CASC-J4 was held on 13th August 2008, as part of the 4th International Joint Conference on Automated Reasoning[1], in Sydney, Australia. It was the thirteenth competition in the CASC series (lucky for some). The CASC-J4 web site provides access to details of the competition design, competition resources, and the systems that were entered: `http://www.tptp.org/CASC/J4/`

CASC-J4 was (like all CASCs) divided into divisions according to problem and system characteristics. There were competition divisions in which systems were explicitly ranked, and a demonstration division in which systems could demonstrate their abilities without being formally ranked. For analysis purposes, the divisions were additionally split into categories according to the syntactic characteristics of the problems. The competition divisions were (see the web site for details):

– FOF - First-order form theorems (axioms with a provable conjecture)
– FNT - FOF non-theorems (axioms with a counter-satisfiable conjecture, and satisfiable axiom sets)
– CNF - Clause normal form theorems (unsatisfiable clause sets)
– SAT - CNF non-theorems (satisfiable clause sets)
– EPR - CNF effectively propositional theorems and non-theorems
– UEQ - Unit equality CNF theorems
– LTB - FOF problems from large theories, presented in batches.

All the divisions had an assurance ranking class, in which the systems were ranked in each division according to the number of problems solved (just a "yes" output). The FOF, FNT, and LTB divisions additionally had a proof/model ranking class in which the systems were ranked according to the number of problems solved with an acceptable proof/model output. The problems were

[1] CADE was a constituent conference of IJCAR, hence CASC-"J4".

selected from the TPTP problem library v3.5.0, which was not released until the day of the competition. The selection of problems was biased towards up to 50% new problems not previously seen by the entrants. A CPU time limit was imposed on each system's run on each problem, and a wall clock time limit was imposed to limit very high memory usage that causes swapping.

The main change in CASC-J4 since CASC-21 (the previous CASC) was the addition of the LTB division. Each category of this division uses a theory in which there are many functors and predicates, many axioms of which typically only a few are required for the proof of a theorem, and many theorems to be proved using a common core set of axioms. The problems of each category are provided to the ATP systems as a batch, allowing the ATP systems to load and preprocess the common core set of axioms just once, and to share logical and control results between proof searches. Articulate Software provided $3000 of prize money for the SMO category of the LTB division. Minor changes since CASC-21 were: the discontinuation of the CNF proof class and SAT model class; a requirement that systems' output differentiate between theorems, non-theorems, unsatisfiable formula sets, and satisfiable formula sets; a requirement that systems' output differentiate between proofs and models; and explicitly encouragement that entered systems have an open source license.

CASC has been a catalyst for improved engineering and performance of many ATP systems. Over the years the CASC design has been incrementally adapted to the needs of the growing number of "real world" users of ATP systems. Contributions to the TPTP problem library provide insights into the types of problems that users need to solve, which in turn feedback into changes to CASC. The increased emphasis on FOF (rather than CNF) problems, and the addition of the LTB division, are consequences of this process. Users and developers of ATP systems are strongly encouraged to continue contributing new problems to the TPTP, to help ensure that CASC and general ATP development efforts are focussed in the right direction.

Labelled Splitting

Arnaud Fietzke and Christoph Weidenbach

Max-Planck-Institut für Informatik, Campus E1 4
D-66123 Saarbrücken
{fietzke,weidenbach}@mpi-inf.mpg.de

Abstract. We define a superposition calculus with explicit splitting and an explicit, new backtracking rule on the basis of labelled clauses. For the first time we show a superposition calculus with explicit backtracking rule sound and complete. The new backtracking rule advances backtracking with branch condensing known from Spass. An experimental evaluation of an implementation of the new rule shows that it improves considerably the previous Spass splitting implementation. Finally, we discuss the relationship between labelled first-order splitting and DPLL style splitting with intelligent backtracking and clause learning.

1 Introduction

Splitting is an inference rule for case analysis. It is well-known from the Davis-Putnam-Logemann-Loveland (DPLL) [1] decision procedure for propositional logic, where a propositional clause set N is split into the clause sets $N \cup \{A\}$ and $N \cup \{\neg A\}$ for some propositional variable A occurring in N. Obviously, N is satisfiable iff one of the two split clause sets is satisfiable. Furthermore, both split clause sets are simpler than N in the sense that any clause from N containing A can be removed, and in all other clauses from N any occurrence of $\neg A$ can be removed for the split clause set $N \cup \{A\}$[1]. The DPLL decision procedure does not consider the two split clause sets in parallel by duplicating N, but traverses the eventual tree generated by splitting and backtracking in a depth-first way. By appropriate implementation mechanisms, backtracking then becomes quite cheap. As any split set is a subset of subclauses after reduction with the split variable, updates can be made my marking and there is no need to generate new clause objects[2]. Nieuwenhuis et al. [2] presented the DPLL procedure performing depth-first search by an abstract calculus. One contribution in this paper is to perform the same exercise for the first-order case and splitting.

In first-order logic the DPLL splitting style does typically not make sense, because for a given first-order atom A there exist infinitely many ground instances $A\sigma$ of A and it is not known which instances eventually contribute to a proof or model. Furthermore, in case of models having infinite domains, such a style of splitting won't terminate. Therefore, for many superposition based decidability

[1] Accordingly for $N \cup \{\neg A\}$.

[2] Learning clauses is a separate issue.

results of first-order logic fragments, e.g., [3], a different style of splitting is used. Given a clause $C \in N$ that can be decomposed into two non-trivial variable disjoint subclauses C_1, C_2, we split into the clause sets $N \cup \{C_1\}$ and $N \cup \{C_2\}$. Very often the rule is further restricted to require that both C_1 and C_2 contain at least one positive literal, i.e., we split into clause sets that are closer to Horn. The rationale behind this restriction is that for Horn clause sets decidability results are typically "easier" to establish and more efficient algorithms exist. For example, satisfiability of propositional Horn clause sets can be decided in linear time, whereas satisfiability of arbitrary propositional clause sets is an NP-complete problem.

A further major difference between first-order splitting and propositional splitting is that in the first-order case effective theorem proving typically relies on the generation of new clauses, either via inferences or reductions. Therefore, the bookkeeping for backtracking of a depth-first approach to splitting gets more involved, because marking algorithms on existing clauses are no longer sufficient to track changes. We need to extend the labels used in the abstract DPLL calculus that are sequences of decision and propagation literals, to a sequence of split elements, called the split stack, holding in particular the potential second part of the split clause and all clauses that became redundant in the course of splitting and may have to be reactivated.

Our starting point is the splitting approach as it was implemented in Spass [4,5]. On the basis of this calculus we develop the labelled splitting calculus that in particular refines the previous one with an improved backtracking rule (Section 2). We show the labelled splitting calculus to be sound and complete where we introduce a new notion of fairness, taking into account an infinite path in the split tree. Labelled splitting is implemented in Spass (`http://spass-prover.org/`) and improves significantly on the previous implementation (Section 3). We compare the calculus to other approaches to splitting, in particular, to the DPLL approach with intelligent backtracking and clause learning (Section 3). The paper ends with a summary of the achieved results and directions for future work (Section 4). Missing proofs and further technical details can be found in a technical report [6].

2 Labelled Splitting

We employ the usual notions and notations of first-order logic and superposition in a way consistent with [5]. When traversing the split tree, the conclusions of splits that were used in deriving a clause determine the clause's scope in the tree, i.e., those parts of the tree in which the clause may participate in proof search. In order to capture this information, we identify each case of a splitting application on a given path through the tree with a unique integer, its *split level*, and label each clause with a set of integers, representing all splits that contributed to the derivation of the clause.

Formally, a labelled clause $L : C$ consists of a finite set $L \subseteq \mathbb{N}$ and a clause $C = \Gamma \rightarrow \Delta$ where Γ and Δ contain the negatively and positively occurring atoms, respectively. The empty clause with label L is denoted by $L\colon \Box$. We call the greatest element in L the *split level* of the clause. We say that $L : C$ *depends* on l if $l \in L$. We extend the usual notions about clause sets to sets of labelled clauses in the natural way: for example, we will say that a set N of labelled clauses entails some formula ϕ (written $N \models \phi$) if and only if the corresponding set of unlabelled clauses entails ϕ. Similarly we extend clause orderings [5] to labelled clauses by abstracting from labels.

A *labelled clause set* is of the form $\Psi : N$ where N is a set of labelled clauses and Ψ is the *split stack*. Split stacks are sequences $\Psi = \langle \psi_n, \ldots, \psi_1 \rangle$ of length $n \geq 0$ ($\Psi = \langle\rangle$ if $n = 0$) and correspond to paths in the split tree. The ψ_i are tuples $\psi_i = (l_i, B_i, D_i, \varphi_i)$ called *splits*, where $l_i \in \mathbb{N}$ is the split level, B_i is the set of *blocked* clauses, D_i is the set of *deleted* clauses and $\varphi_i \in \{\varnothing, \{L\}\}$ with $L \subseteq \mathbb{N}$, is the *leaf marker* which records which splits were involved in refuting a branch. This information is exploited during backtracking to get rid of unnecessary splits. Splitting a clause results in a new split being put onto the stack, which can be thought of as entering the corresponding left branch of the split tree. Once the branch has been refuted, the split is removed and possibly replaced by a split representing the right branch of the splitting step. Splits corresponding to left branches will be assigned odd levels, while splits corresponding to right branches will have even levels. Therefore we define two predicates, left and right, as follows: $\mathrm{left}(l) = \mathrm{true}$ iff $l \bmod 2 = 1$ and $\mathrm{right}(l) = \mathrm{true}$ iff $l \bmod 2 = 0$. We call $\psi = (l, B, D, \varphi)$ a *left split* if $\mathrm{left}(l)$, and a *right split* otherwise. In a left split, the set B will contain clauses to be reinserted when entering the right branch. In the present framework, B will always consist of just the second split clause. However using a set allows for additional clauses to be added to the right branch, for example the negation of the first split clause in case it is ground. Furthermore, the reason why split levels are made explicit (instead of taking the split level of ψ_k to be k, for example) is that because of branch condensation, split removal during backtracking is not limited to toplevel splits and hence "holes" may appear in the sequence of split levels. This will become clear when we discuss backtracking. For better readability, we will use the notation $\psi[x := v]$ where x is one of l, B, D, φ to denote a split identical to ψ up to component x, which has value v. We write $\psi[x_1 := v_1, x_2 := v_2]$ instead of $\psi[x_1 := v_1][x_2 := v_2]$.

For the definition of the labelled calculus we distinguish inference rules

$$\mathcal{I}\frac{\Psi : N \quad L_1\colon \Gamma_1 \to \Delta_1 \quad \ldots \quad L_n\colon \Gamma_n \to \Delta_n}{\Psi' : N' \qquad\qquad K\colon \Pi \to \Lambda}$$

and reduction rules

$$\mathcal{R}\frac{\Psi : N \quad L_1\colon \Gamma_1 \to \Delta_1 \quad \ldots \quad L_n\colon \Gamma_n \to \Delta_n}{\Psi' : N' \qquad\qquad \begin{array}{c} K_1\colon \Pi_1 \to \Lambda_1 \\ \vdots \\ K_k\colon \Pi_k \to \Lambda_k \end{array}}$$

The clauses $L_i\colon \Gamma_i \to \Delta_i$ are called the *premises* and the clauses $K_{(i)}\colon \Pi_{(i)} \to \Lambda_{(i)}$ the *conclusions* of the respective rule. A rule is applicable to a labelled clause set

$\Psi : N$ if the premises of the rule are contained in N. In the case of an inference, the resulting labelled clause set is $\Psi' : (N' \cup \{K\colon \Pi \to \Lambda\})$. In the case of a reduction, the resulting labelled clause set is $\Psi' : (N' \setminus \{L_i\colon \Gamma_i \to \Delta_i \mid 1 \leq i \leq n\} \cup \{K_j\colon \Pi_j \to \Lambda_j \mid 1 \leq j \leq k\})$. Furthermore, we say that a clause C is *redundant* in N if it follows from smaller clauses in N, and an inference is redundant in N, if its conclusion follows logically from clauses in N that are smaller than its maximal premise.

We present a basic set of inference and reduction rules which together yield a sound and refutationally complete calculus for first-order logic without equality. The emphasis of this paper lies on the modelling of the splitting process, hence more advanced rules like those discussed in [5] have been omitted, since their presentation here would not add to the understanding of splitting-specific issues. Such rules can be integrated to the present setting in a straightforward way and are contained in our implementation. For the same reason we omit the usual ordering restrictions and selection strategies.

Definition 1 (Splitting). *The inference*

$$\mathcal{I}\frac{\Psi : N \qquad L\colon \Gamma_1, \Gamma_2 \to \Delta_1, \Delta_2}{\Psi' : N \qquad\quad L'\colon \Gamma_1 \to \Delta_1}$$

where $\Psi = \langle \psi_n, \ldots, \psi_1 \rangle$ ($l_n = 0$ *if* $n = 0$), $l_{n+1} = 2\left\lceil \frac{l_n}{2} \right\rceil + 1$, $L' = L \cup \{l_{n+1}\}$, $L'' = L \cup \{l_{n+1} + 1\}$, $\Psi' = \langle \psi_{n+1}, \psi_n, \ldots, \psi_1 \rangle$ *with* $\psi_{n+1} = (l_{n+1}, \{L''\colon \Gamma_2 \to \Delta_2\}, \emptyset, \emptyset)$, $vars(\Gamma_1 \to \Delta_1) \cap vars(\Gamma_2 \to \Delta_2) = \emptyset$, *and* $\Delta_1 \neq \emptyset$ *and* $\Delta_2 \neq \emptyset$ *is called* splitting.

Splitting creates a new split representing the left branch $\Gamma_1 \to \Delta_1$ on the stack. The remainder is kept in the new split's blocked clause set to be restored upon backtracking. The split level l_{n+1} is the smallest odd number larger than l_n (hence left(l_{n+1}) holds) and the blocked clause has $l_{n+1} + 1$ added to its label (right$(l_{n+1} + 1)$ holds). Furthermore, note that splitting is an inference and the parent clause $\Gamma_1, \Delta_1 \to \Gamma_2, \Delta_2$ is not removed from the clause set. A concrete proof strategy may require to apply subsumption deletion to the parent clause immediately after each splitting step (and after each backtracking step, when the corresponding right branch was entered), thus turning splitting into a reduction. In the current Spass implementation, splitting is a reduction rule in this sense.

In case the first split part $L'\colon \Gamma_1 \to \Delta_1$ is ground, the clauses resulting from its negation can be added to the set of blocked clauses, i.e., the right branch. With this modification, the above splitting rule is as powerful as the DPLL splitting rule concerning proof complexity.

Definition 2 (Resolution). *The inference*

$$\mathcal{I}\frac{\Psi : N \quad L_1\colon \Gamma_1 \to \Delta_1, A \quad L_2\colon \Gamma_2, B \to \Delta_2}{\Psi : N \qquad L_1 \cup L_2\colon (\Gamma_1, \Gamma_2 \to \Delta_1, \Delta_2)\sigma}$$

where σ *is the most general unifier of* A *and* B *is called* resolution.

Definition 3 (Factoring). *The inference*

$$\mathcal{I}\frac{\Psi : N \quad L\colon \Gamma \to \Delta, A, B}{\Psi : N \quad L\colon (\Gamma \to \Delta, A)\sigma}$$

where σ is the most general unifier of A and B is called factoring.

Definition 4 (Subsumption Deletion). *The reduction*

$$\mathcal{R}\frac{\Psi : N \quad L_1\colon \Gamma_1 \to \Delta_1 \quad L_2\colon \Gamma_2 \to \Delta_2}{\Psi' : N \qquad L_1\colon \Gamma_1 \to \Delta_1}$$

where $\Gamma_2 \to \Delta_2$ is subsumed by $\Gamma_1 \to \Delta_1$, $l_{m_1} = max(L_1)$, $l_{m_2} = max(L_2)$, $\Psi = \langle \psi_n, \ldots, \psi_{m_1}, \ldots \psi_1 \rangle$, and

$$\Psi' = \begin{cases} \Psi & \text{if } m_1 = m_2 \\ \langle \psi_n, \ldots, \psi_{m_1}[D := D_{m_1} \cup \{L_2\colon \Gamma_2 \to \Delta_2\}], \ldots, \psi_1 \rangle & \text{otherwise} \end{cases}$$

is called subsumption deletion.

The subsumption deletion rule is presented here as one prominent example of a reduction rule, of which many more exist [5]. They all have in common that a clause is simplified (or removed), either because of some property inherent to the clause itself (e.g., tautology deletion), or because of the presence of another clause (as with subsumption deletion). In the first case, no particular precautions are needed with respect to splitting. In the second case however, we must account for the possibility that the reducing (here: subsuming) clause will eventually be removed from the clause set, e.g., because a split that the clause depends on is removed during backtracking. This is why we store the subsumed clause at the subsuming clause's level on the split stack. As an example, consider the clauses $\{1, 6\} : P(x)$ and $\{1, 3\} : P(a)$. Applying subsumption deletion would cause $\{1, 3\} : P(a)$ to be removed from the clause set and added to the deleted set at the split with level 6. On the other hand, if both the subsumer and the subsumee have the same split level, then there always remains a subsuming clause in the current clause set as long as the corresponding split is not deleted, hence we can remove and forget the subsumed clause.

We will now define rules that formalize backtracking. In particular, we focus our attention on the deletion of splits from the split stack to ensure that all the bookkeeping is done correctly. We denote by $\mathrm{maxr}(\Psi) := max(\{1 \le i \le n \mid \mathrm{right}(l_i)\} \cup \{0\})$ the last right split in Ψ. For any given split stack Ψ, we define the set $levels(\Psi)$ to be the set of split levels occurring in Ψ, i.e., $levels(\Psi) := \{l_1, \ldots, l_n\}$ for $\Psi = \langle \psi_n, \ldots, \psi_1 \rangle$. Finally, for any labelled clause set N and set of split levels $K \subseteq \mathbb{N}$ we define $N|_K := \{L\colon \Gamma \to \Delta \in N \mid L \subseteq K\}$.

When removing a split ψ_k from the stack, we have to take care to undo all reductions that involved a clause depending on (the level of) ψ_k. In particular, if a clause C_1 depending on ψ_k was used to reduce some other clause C_2, then C_2 must be reinserted into the current clause set. The reason is that C_1 will be removed from the current clause set, and C_2 may then no longer be redundant. If C_2 was reduced by C_1, then C_2 will be in the set of deleted clauses at the level of C_1. Note that although we know that C_1 depends on ψ_k, C_1 may also depend on other splits and thus have a split level higher than l_k. Our goal is to reinsert the deleted clauses at C_1's split level. But C_1 itself, after having reduced C_2, may have been reduced by some clause C_3, hence C_1 will not necessarily be in the current clause set, but in the deleted set at the level of C_3. This means that we need to reinsert all deleted clauses from split levels of clauses depending on ψ_k. So let $\Psi : N$ be an arbitrary labelled clause set with Ψ of length n, and $1 \leq k \leq n$. Now define

$$D(k) := \bigcup_{\substack{i=1\\ i\neq k}}^{n} D_i \quad \text{and} \quad R(k) := \{j \mid L\colon C \in N \cup D(k),\ l_j = max(L),\ l_k \in L\}.$$

The set $R(k)$ describes the splits corresponding to all levels of clauses that depend on ψ_k, both in N and any deleted set D_i. It follows that the set $\bigcup_{j\in R(k)} D_j$ contains all clauses that may have been reduced by a clause in N depending on ψ_k. The reason for excluding D_k from $D(k)$ is that D_k will always be reinserted when deleting ψ_k, as the following definition shows:

Definition 5 (Delete Split). *We define* $delete(\Psi : N, k) := \Psi' : N'$ *where* $\Psi' = \langle \psi'_n, \ldots, \psi'_{k+1}, \psi'_{k-1}, \ldots, \psi'_1 \rangle$, *and* $N' = (N \cup D_k \cup \bigcup_{j\in R(k)} D_j)|_{levels(\Psi')}$ *with*

$$\psi'_j = \begin{cases} \psi_j[D := \varnothing] & \textit{if } j \in R(k) \\ \psi_j[D := \{L\colon \Gamma \rightarrow \Delta \in D_j \mid l_k \notin L\}] & \textit{otherwise} \end{cases}$$

which removes split ψ_k, all clauses depending on ψ_k, and reinserts all clauses reduced by a clause depending on split ψ_k.

Note that reinserting all clauses in D_j with $l_j \in R(k)$ is an over-approximation, since not every clause in D_j was necessarily reduced by a clause depending on ψ_k. In fact, it may well be that no clause in D_j was reduced by a clause depending on ψ_k. If we wanted to reinsert only clauses reduced by clauses depending on ψ_k, we would have to record which clause was used in each reduction step, not only the reducing clause's split level. It is not clear whether that additional effort would pay off in practice.

We now define a reduction[3] relation $\Psi : N \rightarrow \Psi' : N'$ on labelled clause sets to capture the structural transformations of the stack taking place during backtracking. The reduction relation is defined by the following four rules, where we assume that $\Psi : N$ is a labelled clause set and Ψ is of length n.

Definition 6 (Backjump). *If* $n > 0$, $L\colon \square \in N$ *and* $\max(L) < l_n$, *then*

$$\Psi : N \rightarrow delete(\Psi : N, n).$$

Definition 7 (Branch-condense). *If* $n > 0$, $L\colon \square \in N$, $\max(L) = l_n$, $\text{left}(l_n)$ *and* $k_{max} := \max\ \{k \mid \text{maxr}(\Psi) < k \leq n \textit{ and } l_k \notin L\}$ *exists, then*

$$\Psi : N \rightarrow delete(\Psi : N, k_{max}) \qquad .$$

[3] Not to be confused with reduction rules for clause sets. See [7] for a discussion of abstract reduction systems.

Definition 8 (Right-collapse). *If* $n > 0$, $L_2 \colon \Box \in N$, $\max(L_2) = l_n$ *and* $\mathrm{right}(l_n)$, *and* $\varphi_n = \{L_1\}$, *then*

$$\Psi : N \to \Psi' : (N' \cup \{L \colon \Box\}),$$

where $\Psi' : N' = delete(\Psi : N, n)$ *and* $L = L_1 \cup L_2 \setminus \{l_n - 1, l_n\}$.

Definition 9 (Enter-right). *If* $n > 0$, $L \colon \Box \in N$, $\max(L) = l_n$, $\mathrm{left}(l_n)$ *and* $l_k \in L$ *for all* k *with* $\mathrm{maxr}(\Psi) < k \leq n$, *then* $\Psi'' : N'' := delete(\Psi : N, n)$ *and*

$$\Psi : N \to \Psi' : N',$$

where $\Psi' = \langle (l_n + 1, \emptyset, \emptyset, \{L\}), \psi''_{n-1}, \ldots, \psi''_1 \rangle$ *and* $N' = N'' \cup B_n$.

Note that at most one rule is applicable to any given labelled clause set containing one empty clause, since the preconditions are mutually exclusive. Furthermore, the length of the split stack together with the number of empty clauses in a labelled clause set induce a well-founded partial ordering on labelled clause sets, with respect to which each rule is decreasing. Hence any sequence $\Psi : N \to \Psi' : N' \to \ldots$ terminates and each labelled clause set $\Psi : N$ has a unique normal form with respect to $\to$, which we write $\Psi : N{\downarrow}$. We are now ready to give the definition of the backtracking rule:

Definition 10 (Backtracking). *The reduction*

$$\mathcal{R}\frac{\Psi : N \qquad L \colon \Box}{(\Psi : N'){\downarrow}}$$

where $N' = \{L' \colon C \in N \mid C \neq \Box\} \cup \{L \colon \Box\}$ *is called* backtracking.

Since we have not placed any restrictions on when to apply backtracking, there may be more than one empty clause in $\Psi : N$. Choosing one and removing all others before applying stack reductions ensures that the result of backtracking is uniquely defined. In a practical system, one would typically choose the most general empty clause for backtracking, i.e., the one whose label represents a maximal scope in the split tree.

The stack reduction rule that improves upon previous backracking mechanisms is Right-collapse, which analyzes the dependencies involved in refuting left and right branches and computes a newly labelled empty clause by removing complementary split levels. This allows consecutive sequences of Backjump steps (interleaved by applications of Right-collapse) to take place within a single Backtracking step, thus possibly pruning larger parts of the split tree. The following example shows the stack reduction rules in action.

Example 1. Consider the clause set

$$\begin{array}{lll}\{ \to P(a), Q(b); & P(x) \to P(f(x)), R(c); & P(f(f(a))) \to; \\ \to Q(x), S(y); & Q(x), P(x) \to; & \\ \to R(a), R(b); & S(x), P(x) \to \} & \end{array}$$

where we omit empty labels. Figure 1 shows the development of the split tree over three backtracking steps. Bold line segments indicate the current path in the split tree, and numbers next to branches correspond to split levels. The split stack representing the first tree is

$$\langle\ (7, \{\{8\}\colon R(c)\}, \varnothing, \varnothing), (5, \{\{6\}\colon R(b)\}, \varnothing, \varnothing), \\ (3, \{\{4\}\colon R(c)\}, \varnothing, \varnothing), (1, \{\{2\}\colon Q(b)\}, \varnothing, \varnothing)\ \rangle$$

which is obtained in seven steps from the initial clause set: (1) clause $\rightarrow P(a), Q(b)$ is split, (2) resolution is applied to $P(a)$ and $P(x) \rightarrow P(f(x)), R(c)$, (3) the resulting clause $\{1\}{:}\rightarrow P(f(a)), R(c)$ is split, (4) clause $\rightarrow R(a), R(b)$ is split, (5) resolution is applied to $\{1,3\}\colon P(f(a))$ and $P(x) \rightarrow P(f(x)), R(c)$, resulting in $\{1,3\}{:}\rightarrow P(f(f(a))), R(c)$, which is again split (6), finally the empty clause $\{1,3,7\}\colon \square$ is derived by resolving $\{1,3,7\} : P(f(f(a)))$ and $P(f(f(a))) \rightarrow$ (7). The third split did not contribute to the contradiction, so it is removed by Branch-condense in step 1, followed by Enter-right, which produces $(8, \varnothing, \varnothing, \{\{1,3,7\}\})$ as toplevel split. The clause $\rightarrow Q(x), S(y)$ is then split, resolution is applied to $\{1\}\colon P(a)$ and $Q(x), P(x) \rightarrow$ to derive $\{1\}\colon Q(x) \rightarrow$, which is resolved with $\{9\} : Q(x)$ to yield $\{1,9\}\colon \square$. Enter-right is applied (step 3), producing toplevel split $(10, \varnothing, \varnothing, \{\{1,9\}\})$, and the empty clause $\{1,10\}\colon \square$ is derived using clauses $\{1,10\} : S(y)$ and $S(x), P(x) \rightarrow$ and $\{1\} : P(a)$. In step 4, the clause labels $\{1,9\}$ and $\{1,10\}$ are collapsed into $\{1\}$. Finally, two Backjump steps followed by Enter-right yield the last tree, which corresponds to the split stack $\langle(2, \varnothing, \varnothing, \{\{1\}\})\rangle$. Observe how the empty clause generated in step 4 allows us to skip branch 4 and jump directly to branch 2, which would not be possible without the Right-collapse rule.

We now state soundness and completeness results for the labelled calculus. Proving these is a technically involved exercise, hence in this paper we limit ourselves to the main definitions and theorems, referring to [6] for the details. A *derivation* in the labelled calculus is a sequence of labelled clause sets

$$\mathcal{D} = \Psi_0 : N_0 \ \rhd\ \Psi_1 : N_1 \ \rhd\ \Psi_2 : N_2 \ \rhd\ \ldots$$

such that $\Psi_0 = \langle\rangle$, N_0 is the initial labelled clause set and for each $i \geq 1$, the labelled clause set $\Psi_i : N_i$ is the result of applying a rule of the calculus to $\Psi_{i-1} : N_{i-1}$. We call the step $\Psi_{i-1} : N_{i-1} \ \rhd\ \Psi_i : N_i$ the ith step of $\mathcal{D}$. We write $\Psi : N \rhd^* \Psi' : N'$ to indicate that $\Psi' : N'$ is obtained from $\Psi : N$ by zero or more applications of calculus rules. We use superscripts to make explicit that a split belongs to a particular split stack in a derivation: for example, we write ψ^i_j for the jth split of Ψ_i, and D^i_j for the deleted set of ψ^i_j. In order to prove soundness, we extend the notion of satisfiability to labelled clause sets. Since we are exploring a tree whose branches represent alternatives of successive case distinctions, we associate a clause set with each unexplored branch on the stack. That is, we associate with each *left* split a set of labelled clauses corresponding to the *right* branch of the splitting step. Formally, let $\Psi_i : N_i$ be a labelled clause set in a derivation. We define the following set of clause sets:

$$\mathcal{N}_i := \{(N_i \cup \bigcup_{j=1}^{n} D_j)|_{L_k} \cup B_k \ \mid\ k \in \{1, \ldots, n\}, L_k = \{l_1, \ldots, l_{k-1}\} \text{ and } \mathrm{left}(l_k)\}.$$

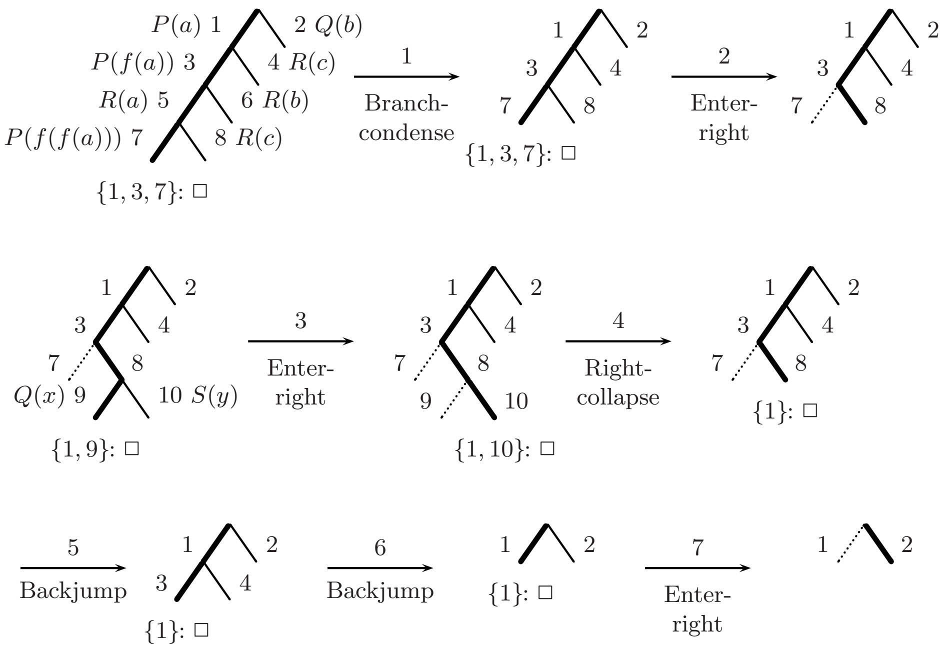


Fig. 1. Split Tree Development over 3 Backtracking Steps

We use the notation N_i^k to denote the set $(N_i \cup \bigcup_{j=1}^n D_j)|_{L_k} \cup B_k \in \mathcal{N}_i$. We call N_i the *active clause set* of $\Psi_i : N_i$, and $\mathcal{N}_i$ the set of *inactive clause sets* of $\Psi_i : N_i$.

Definition 11 (Satisfiability of Labelled Clause Sets). *We say that $\Psi_i : N_i$ is* satisfiable, *if and only if*

- *N_i is satisfiable, or*
- *some $N_i^k \in \mathcal{N}_i$ is satisfiable.*

It can now be shown that every rule of the labelled calculus preserves satisfiability of labelled clause sets, where the proof of the backtracking case is handled by induction over the stack reduction relation. Among the derivation invariants needed for the proof are *label validity* (clause labels only refer to existing splits in the stack) and *path validity* (all clauses in the active and deleted sets follow logically from initial clauses and split clauses on the current path, and initial clauses together with split clauses described by leaf markers are unsatisfiable).

Theorem 1 (Soundness). *Let N be an arbitrary clause set. Let $N_0 := \{\varnothing : C \mid C \in N\}$ be the associated set of labelled clauses, let $\Psi_0 := \langle\rangle$, and let $\Psi_0 : N_0 \rhd^* \Psi_m : N_m$ be an arbitrary derivation starting with $\Psi_0 : N_0$. Then $\Psi_m : N_m$ is satisfiable if and only if $\Psi_0 : N_0$ is satisfiable.*

When backtracking is modelled explicitly, as we have done here, classical model construction techniques (as in [8,9]) cannot directly be used to show completeness of the calculus. In particular, defining a fair derivation to be a derivation in which any non-redundant inference from persistent clauses is eventually computed, no longer guarantees that any fair derivation from an unsatisfiable clause set eventually yields a contradiction. The reason is that in our calculus, the changes to the clause set are not monotonic (in the sense that in each step, only derived clauses are added or redundant clauses are removed).

For example, consider the unsatisfiable clause set

$$\begin{aligned} N_0 = \{\ & S(a); \qquad \neg S(a); \qquad P(a);\\ & P(x) \to Q(y), P(f(x));\\ & Q(y) \to S(x) \qquad\qquad\qquad \} \end{aligned}$$

where all clauses have the empty label. We construct an infinite derivation $\mathcal{D} = \Psi_0 : N_0 \rhd \Psi_1 : N_1 \rhd \ldots$ as follows: we apply resolution to derive a clause $\to Q(y), P(f^{n+1}(a))$ (initially, $n = 0$) which we then split. In the left branch, we use $Q(y)$ to infer $S(x)$. We then apply subsumption deletion to $S(x)$ and $S(a)$, removing $S(a)$ from the clause set and storing it in the deleted set of the current split. We apply resolution to infer the empty clause from $S(x)$ and $\neg S(a)$ and backtrack, entering the right branch, reinserting clause $S(a)$. We then repeat the procedure with n increased by one (see Figure 2). The classical definition of persistent clauses as $N_\infty = \bigcup_i \bigcap_{j \geq i} N_j$ yields a set that does not contain $S(a)$ (thus N_∞ is satisfiable), because for each subsumption deletion step k, we have $S(a) \notin N_k$, hence $S(a)$ is not "persistent" when viewed over the whole derivation.

$$\begin{tikzcd}
& \dfrac{P(f^n(a)) \quad P(x) \to Q(y) \vee P(f(x))}{Q(y) \vee P(f^{n+1}(a))} \arrow[dl] \arrow[dr] & \\
\dfrac{Q(y) \quad Q(y) \to S(x)}{S(x)} \arrow[d, phantom, "\vdots"] & & \ddots \\
\Box & &
\end{tikzcd}$$

Fig. 2. Split Tree Development

In the above example, the non-persistent clause $S(a)$ is redundant in all left branches. However, every left branch is refuted at some point, causing $S(a)$ to be reinserted upon backtracking. Thus, the fact that the clause is redundant only in branches that are eventually closed is irrelevant in an infinite derivation. In the example, the split tree has an infinite path consisting of right branches, and along this path, the clause $S(a)$ is not redundant and should therefore eventually be considered for inferences. Our goal is therefore to define a notion of fairness that ignores closed branches and only talks about clauses on the infinite path of the split tree. In the following, we use $\mathrm{suf}_{\mathcal{D}}(i)$ to denote the suffix $\Psi_i : N_i \rhd \Psi_{i+1} : N_{i+1} \rhd \ldots$ of an infinite derivation $\mathcal{D} = \Psi_0 : N_0 \rhd \Psi_1 : N_1 \rhd \ldots$.

Definition 12 (Persistent Split, Weakly Persistent Clause). *Given an infinite derivation*

$$\mathcal{D} = \Psi_0 : N_0 \rhd \Psi_1 : N_1 \rhd \ldots,$$

and $i \geq 1$, where $\Psi_i = \langle \psi_{n_i}, \ldots, \psi_1 \rangle$, we call ψ_k $(1 \leq k \leq n_i)$ persistent *in* $\mathrm{suf}_{\mathcal{D}}(i)$*, if $l_k \in \mathrm{levels}(\Psi_j)$ for all $j \geq i$. Furthermore, we call a clause $L : C \in \left(N_i \cup \bigcup_{j=1}^{n_i} D_j^i\right)$* weakly persistent *in* $\mathrm{suf}_{\mathcal{D}}(i)$*, if for all $l_j \in L$, ψ_j is persistent in* $\mathrm{suf}_{\mathcal{D}}(i)$.

Weakly persistent clauses are not necessarily contained in every clause set of the derivation from some point on. However, if a clause is weakly persistent, then from some point on, every clause set contains either the clause itself, or some clause that subsumes it. Also note that any clause that is weakly persistent in $\mathcal{D}$ is contained in N_0, since no split is persistent in $\mathcal{D}$.

Definition 13 (Persistent Step). *Given an infinite derivation*

$$\mathcal{D} = \Psi_0 : N_0 \rhd \Psi_1 : N_1 \rhd \ldots,$$

we say that step $i \geq 1$ of $\mathcal{D}$ is persistent*, if*

1. *step i is a splitting or backtracking step, $\Psi_i = \langle \psi_{n_i}, \ldots, \psi_1 \rangle$, and ψ_{n_i} is persistent in* $\mathrm{suf}_{\mathcal{D}}(i)$*, or*
2. *step i is an inference or reduction step and all premises of the applied inference or reduction rule are weakly persistent in* $\mathrm{suf}_{\mathcal{D}}(i-1)$.

We will define the limit of an infinite derivation $\mathcal{D}$ to be the derivation obtained by starting from the original labelled clause set and applying exactly the persistent steps of $\mathcal{D}$. Note that all left splits are created by splitting, whereas all right splits are created by backtracking. The limit will contain no more applications of backtracking. Instead, whenever a persistent right branch is created in the original derivation, the limit will enter that branch directly, using the rule *splitting right*:

Definition 14 (Splitting Right). *The inference*

$$\mathcal{I}\,\frac{\Psi : N \qquad L\colon \Gamma_1, \Gamma_2 \to \Delta_1, \Delta_2}{\Psi' : N \qquad\quad L'\colon \Gamma_2 \to \Delta_2}$$

where $\Psi = \langle \psi_n, \ldots, \psi_1 \rangle$, $L' = L \cup \{l_n + 1\}$, $\Psi' = \langle \psi_{n+1}, \psi_n, \ldots, \psi_1 \rangle$ with $\psi_{n+1} = (l_n + 1, \emptyset, \emptyset, \emptyset)$, $vars(\Gamma_1 \to \Delta_1) \cap vars(\Gamma_2 \to \Delta_2) = \emptyset$, and $\Delta_1 \neq \emptyset$ and $\Delta_2 \neq \emptyset$ is called splitting right.

Definition 15 (Limit of a Derivation). *Given an infinite derivation*

$$\mathcal{D} = \Psi_0 : N_0 \rhd \Psi_1 : N_1 \rhd \ldots,$$

let $i \geq 0$ be a backtracking step, let l be the split level of the toplevel split of Ψ_i, and let $k < i$ be maximal such that step k of $\mathcal{D}$ is a splitting step producing a split of level $l - 1$. Then we define $\mathrm{spt}_{\mathcal{D}}(i) := k$*, the* associated splitting step *of* *backtracking step i. Furthermore, we define the monotonic function $f_{\mathcal{D}} : \mathbb{N} \to \mathbb{N}$ as follows:*

$$\begin{aligned} f_{\mathcal{D}}(0) &:= 0 \\ f_{\mathcal{D}}(i+1) &:= \min\{j > f_{\mathcal{D}}(i) \mid \textit{step } j \textit{ of } \mathcal{D} \textit{ is persistent}\}. \end{aligned}$$

The limit *of $\mathcal{D}$ is defined as*

$$\lim(\mathcal{D}) := \Psi'_0 : N'_0 \rhd \Psi'_1 : N'_1 \rhd \Psi'_2 : N'_2 \rhd \dots$$

where $\Psi'_0 : N'_0 = \Psi_0 : N_0$ and $\Psi'_{i+1} : N'_{i+1}$ is obtained from $\Psi'_i : N'_i$ as follows:

1. *if step $f_{\mathcal{D}}(i+1)$ of $\mathcal{D}$ is a backtracking step: let $(l, \emptyset, \emptyset, M)$ be the toplevel split of Ψ_{i+1}, and consider the associated splitting step $k = \mathrm{spt}_{\mathcal{D}}(i+1)$*

$$\mathcal{I}\frac{\Psi_{k-1} : N_{k-1} \qquad L\colon \Gamma_1, \Gamma_2 \to \Delta_1, \Delta_2}{\Psi_k : N_k \qquad L'\colon \Gamma_1 \to \Delta_1}$$

   Then step $i+1$ of $\lim(\mathcal{D})$ is the splitting right step

$$\mathcal{I}\frac{\Psi'_i : N'_i \qquad L\colon \Gamma_1, \Gamma_2 \to \Delta_1, \Delta_2}{\Psi'_{i+1} : N'_{i+1} \qquad M\colon \Gamma_2 \to \Delta_2}$$

2. *otherwise: $\Psi'_{i+1} : N'_{i+1}$ is obtained by applying the same rule as in step $f_{\mathcal{D}}(i+1)$ of $\mathcal{D}$ to $\Psi'_i : N'_i$.*

Note that in general, neither $N'_i \subseteq N_{f_{\mathcal{D}}(i)}$ nor $N'_i \supseteq N_{f_{\mathcal{D}}(i)}$ hold for arbitrary i, because $N_{f_{\mathcal{D}}(i)}$ may contain clauses that are not (weakly) persistent, and persistent clauses may have been subsumed by non-persistent ones. Therefore we can not simply take the limit to be a subsequence of the initial derivation.

Lemma 1. *For every infinite derivation $\mathcal{D} = \Psi_0 : N_0 \rhd \Psi_1 : N_1 \rhd \dots$, and every $i \geq 1$, if step i of $\lim(\mathcal{D})$ is an inference step, then its conclusion is contained in $N_{f_{\mathcal{D}}(i)}$.*

Proof. Follows directly from Definition 15.

For $\lim(\mathcal{D}) = \Psi'_0 : N'_0 \rhd \Psi'_1 : N'_1 \rhd \dots$ we define $N^{\lim(\mathcal{D})}_\infty := \bigcup_i \bigcap_{j \geq i} N'_j$.

Definition 16 (Fairness). *A derivation*

$$\mathcal{D} = \Psi_0 : N_0 \rhd \Psi_1 : N_1 \rhd \dots$$

is called fair *if*

1. *either $\mathcal{D}$ is finite and in the final clause set N_k all resolution and factoring inferences are redundant in N_k, or $\mathcal{D}$ is infinite and every resolution or factoring inference from $N^{\lim(\mathcal{D})}_\infty$ is redundant in some $N'_i \in \lim(\mathcal{D})$; and*
2. *for every $i \geq 0$ with $L\colon \Box \in N_i$ with $L \neq \emptyset$, there exists $j > i$ such that step j of $\mathcal{D}$ is a backtracking step with premise $L\colon \Box$.*

Lemma 2. *For any fair infinite derivation $\mathcal{D}$ and any $i \geq 0$, no $L\colon \Box$ with $L \neq \varnothing$ is weakly persistent in* $\mathrm{suf}_{\mathcal{D}}(i)$.

Proof. Assume $L \neq \varnothing$ and consider the backtracking step with $L\colon \Box$ as premise (which must exist by condition 2 of Definition 16). It follows from the definitions of the stack reduction rules that the final rule applied is Enter-right, which deletes the split with the greatest split level in L.

Theorem 2 (Completeness). *Let $\mathcal{D} = \Psi_0 : N_0 \rhd \Psi_1 : N_1 \rhd \ldots$ be a fair derivation. If $\varnothing\colon \Box \notin N_\infty^{\lim(\mathcal{D})}$, then N_0 is satisfiable.*

The proof of Theorem 2 closely follows the one given in [8] (Theorem 4.9), with the exception that we limit ourselves to showing that unsatisfiability of N_0 implies $\varnothing\colon \Box \in N_\infty^{\lim(\mathcal{D})}$. The main observation that allows us to use the framework from [8] is that splitting (and splitting right) is unsatisfiability-preserving with respect to the sequence $N'_0, N'_1, N'_2, \ldots$ of clause sets (more precisely, their unlabelled equivalents) in $\lim(\mathcal{D})$. Fairness then ensures that the set $N_\infty^{\lim(\mathcal{D})}$ is saturated up to redundancy with respect to resolution and factoring.

3 Experiments and Related Work

3.1 Experiments

Our enhanced backtracking process has been integrated into the Spass [10] theorem prover. The basis for the implementation was Spass version 3.0. As we mentioned before, the calculus presented here represents a minimal set of inference and reduction rules, whereas the overall splitting calculus is implemented in Spass [5], and our extension covers that entire calculus. The data structures used to represent the split stack were modified to allow storage of the dependencies of closed left branches. Minor modifications to the implementation of reduction rules were made to ensure that reduced clauses are always recorded for later reinsertion. These modifications were necessary since the original branch condensation in Spass is performed only up to the last backtracking level, hence a redundant clause is recorded only if it has been subsumed by a clause with greater split level. Finally, a new backtracking procedure was written, implementing the stack reduction rules.

The implementation was tested on the TPTP problem library, version 3.2.0 [11], which consists of 8984 first-order problems. On 2513 of those problems, Spass (in automatic mode) uses the splitting rule during proof search. For the experiments, an Opteron Linux cluster was used, and Spass was given a time limit of 5 minutes per problem.

Overall, the number of splits performed per problem decreased by about 10% on average when using improved backtracking. In addition, Spass with improved backtracking terminated with a solution for 24 problems (22 proofs, 2 completions) that could not be solved by the version without improved backtracking within the given time limit. The new version looses 4 problems because of the potentially different search space exploration caused by the new backtracking rule. These problems are recovered by an increased time limit.

3.2 Related Work

The paper is an extension of the abstract DPLL calculus of Nieuwenhuis et al. [2] for first-order logic. Due to the need to create new clauses via inferences and reductions, the calculus is more involved.

An alternative approach to case analysis in saturation-based theorem proving, relying on the introduction of special propositional symbols instead of a split stack, was presented in [12]. The main difference to our framework is that when simulating splitting with new propositional symbols, the generated clauses can not be directly used for reductions: for example, splitting the clause $P(x) \vee Q(y)$ in this way produces the clauses $P(x) \vee p$ and $p \rightarrow Q(x)$, where p is a new propositional symbol. Therefore, this type of splitting does not provide the necessary support for the superposition based decidability results on non Horn first-order fragments. On the other hand, in particular if applied to unsatisfiable problems, this alternative approach to splitting can be very useful.

For the general methodology of labelled clauses there is a huge literature, see [13] for an overview. In particular, the use of clause labels to model explicit case analysis was first suggested in [14], which provided a starting point for the work presented here. We refined the abstract labelled splitting rule presented there with explicit backtracking and redundancy handling.

The DPLL procedure [1], which lies at the core of most of today's state-of-the-art boolean satisfiability solvers, has received a great deal of attention in the past years. In particular, a lot of research has gone into improving the backtracking process. Thus it is natural to ask how, in the propositional domain, our backtracking scheme compares to that of modern SAT solvers. Let us first clarify some key differences between our framework and DPLL.

In basic DPLL, when exhaustive unit propagation has neither led to a conflict nor yielded a complete truth assignment, a choice is made by assuming some yet undefined literal to be true (typically a literal occurring in some clause). If this choice leads to a conflict, the truth value of that literal is flipped, hence the two branches of the case analysis are A and $\neg A$ for some propositional variable A. This can be simulated by the refined version of the splitting rule, discussed after Definition 1. The reason why this refinement is restricted to ground split parts is that the negation of universally quantified clauses leads to the introduction of new Skolem constants, and in practice this tends to extend the search space for the second branch. This could in principle be avoided by remembering all ground terms that a variable has been instantiated with. For example, if the left split part was $P(x)$ and a refutation of the branch involved substituting x by both a and $f(b)$ in different subtrees, then $\neg P(a) \vee \neg P(f(b))$ could be used instead of $\neg P(c)$ for some new constant c. Although this approach avoids the introduction of new symbols, tracking all instantiations adds overhead and the fact that the resulting lemmata are again in general non-units diminishes their usefulness for reductions. When dealing with purely propositional problems however, this lemma-generation can be used to simulate DPLL-style case analysis: for any clause $\Gamma \rightarrow \Delta, A$ chose A as the first split part – the lemma $\neg A$ is then available in the second branch.

Backtracking in modern SAT solvers is based on a conflict analysis during which a conflict clause is learned, i.e., added to the clause set. We compared SPASS implementing our new backtracking rule (Definition 10) without learning with MiniSat v1.14 [15] on SAT problems, by manipulating both systems such that they essentially use the same propositional variable order for branching. Out of the current SATLIB (http://www.satlib.org) library we selected about 200 problems that SPASS could solve in a 5 min time limit. On more than 95% of all problems Minisat needs less splits than Spass. Out of these problems, Spass performs three times more splits than MiniSat, on the average. This result suggests that conflict-driven clause learning is in favor of the Spass split backtracking mechanism. So there is potential for exploring this mechanism also for the first-order case. However, again, this requires at least a detailed analysis of the closed branches as split clauses may have been used in several instantiations.

Although our backtracking rule does not include learning of conflict clauses, there are cases where it is superior to the conflict driven clause learning of modern SAT solvers, i.e., a proof requires less splits. This is documented by the 5% of examples where SPASS needed less splits than MiniSat and such examples can also be explicitely constructed [6].

4 Conclusion

We have extended the abstract DPLL calculus by Nieuwenhuis et al. [2] to the full first-order case, called labelled splitting. The calculus is sound and complete. For the completeness result we introduced a new notion of an infinite path to establish fairness.

The calculus is implemented for the full superposition calculus [5] in Spass. It shows a 10% average gain in the number of splits on all TPTP problems where splitting is involved and Spass could decide 24 more problems compared to the previous version.

The fairness notion of Definition 16 does not directly provide an effective strategy for a fair inference selection. In Spass we mainly use two different strategies. For the propositional case we employ exhaustive splitting, because splitting combined with the reduction matching replacement resolution [5] constitutes already a complete calculus. For the first-order case, clauses are selected for inferences with respect to their "weight" composed out of the number of contained symbols and their depth in the derivation. If such a clause can be split and the first split part has a "sufficient" reduction potential for other clauses, splitting is preferred over other inferences.

A comparison of the labelled splitting backtracking mechanism with DPLL style backtracking based on conflict-driven clause learning reveals room for further improvement. However, this is not a straightforward effort, because the negation of a first-order clause is an existentially quantified conjunction of literals that via Skolemization introduces new constants to the proof search. It is well known that the addition of new constants causes an increased complexity of the unsatisfiability problem and if potentially done infinitely often, can even cause completeness issues. So it seems to us that in the case of a conflict, an analysis of the proof and the used first-order terms in the proof is the most promising approach to enhance the presented labelled splitting backtracking mechanism with conflict-driven clause learning. This will be subject of future research.

Acknowledgements. We would like to thank the reviewers for their pertinent comments and suggestions that were of great help in writing this article.

References

1. Davis, M., Putnam, H.: A computing procedure for quantification theory. J. ACM 7, 201–215 (1960)
2. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL. Journal of the ACM 53, 937–977 (2006)
3. Bachmair, L., Ganzinger, H., Waldmann, U.: Superposition with simplification as a decision procedure for the monadic class with equality. In: Gottlob, G., Leitsch, A., Mundici, D. (eds.) KGC 1993. LNCS, vol. 713, pp. 83–96. Springer, Heidelberg (1993)
4. Weidenbach, C., Gaede, B., Rock, G.: Spass & flotter, version 0.42. In: McRobbie, M.A., Slaney, J.K. (eds.) CADE 1996. LNCS, vol. 1104, pp. 141–145. Springer, Heidelberg (1996)
5. Weidenbach, C.: Combining superposition, sorts and splitting. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. 2, pp. 1965–2012. Elsevier, Amsterdam (2001)
6. Fietzke, A., Weidenbach, C.: Labelled splitting. Research Report MPI-I-2008-RG1-001, Max-Planck Institute for Informatics, Saarbruecken, Germany (2008)
7. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1998)
8. Bachmair, L., Ganzinger, H.: Resolution theorem proving. In: Handbook of Automated Reasoning, pp. 19–99 (2001)
9. Nieuwenhuis, R., Rubio, A.: Paramodulation-based theorem proving. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. I, pp. 371–443. Elsevier, Amsterdam (2001)
10. Weidenbach, C., Schmidt, R., Hillenbrand, T., Rusev, R., Topic, D.: System description: SPASS version 3.0. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 514–520. Springer, Heidelberg (2007)
11. Sutcliffe, G., Suttner, C.B.: The TPTP Problem Library: CNF Release v1.2.1. Journal of Automated Reasoning 21(2), 177–203 (1998)
12. Riazanov, A., Voronkov, A.: Splitting without backtracking. In: IJCAI, pp. 611–617 (2001)
13. Basin, D., D'Agostino, M., Gabbay, D.M., Matthews, S., Viganó, L. (eds.): Labelled deduction. Kluwer Academic Publishers, Dordrecht (2000)
14. Lev-Ami, T., Weidenbach, C., Reps, T., Sagiv, M.: Labelled clauses. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 311–327. Springer, Heidelberg (2007)
15. Eén, N., Sörensson, N.: An extensible SAT solver. Theory and Applications of Satisfiability Testing, 502–518 (2004)

Engineering DPLL(T) + Saturation

Leonardo de Moura and Nikolaj Bjørner

Microsoft Research, One Microsoft Way, Redmond, WA, 98074, USA
{leonardo,nbjorner}@microsoft.com

Abstract. Satisfiability Modulo Theories (SMT) solvers have proven highly scalable, efficient and suitable for integrated theory reasoning. The most efficient SMT solvers rely on refutationally incomplete methods for incorporating quantifier reasoning. We describe a calculus and a system that tightly integrates Superposition and DPLL(T). In the calculus, all non-unit ground clauses are delegated to the DPLL(T) core. The integration is tight, dependencies on case splits are tracked as hypotheses in the saturation engine. The hypotheses are discharged during backtracking. The combination is refutationally complete for first-order logic, and its implementation is competitive in performance with E-matching based SMT solvers on problems they are good at.

1 Introduction

SMT solvers based on a DPLL(T) [1] framework have proven highly scalable, efficient and suitable for integrating theory reasoning. However, for numerous applications from program analysis and verification, an integration of decision procedures for the ground fragment is insufficient, as proof obligations often include quantifiers for capturing frame conditions over loops, summarizing auxiliary invariants over heaps, and for supplying axioms of theories that are not already equipped with ground decision procedures. A well known approach for incorporating quantifier reasoning with ground decision procedures is E-matching algorithm used in the Simplify theorem prover [2]. The E-matching algorithm works against an E-graph to instantiate quantified variables. Other state-of-the-art SMT solvers also use E-matching: CVC3 [3], Fx7 [4], Yices [5], and Z3 [6]. Although E-matching is quite effective for some software verification problems, it suffers from several problems: it is not refutationally complete for first-order logic, hints (triggers) are usually required, it is sensitive to the syntactic structure of the formula, and fails to prove formulas that can be easily discharged by saturation based provers.

Equational theorem provers based on Superposition Calculus are strong at reasoning with equalities, universally quantified variables, and Horn clauses. However, these provers do not perform well in the context of software verification [7,3], as they explore a huge search space generated by a large number of axioms, most of which are irrelevant. The typical software verification problem consists of a set of axioms and a big (mostly) ground formula encoding the data and control flow of the program. This formula is usually a deeply nested and-or tree. Quantified formulas nested in this tree can be extracted by naming them with fresh propositional variables. These problems typically yield large sets with huge non-Horn clauses, which are far from the sweet spot for saturation based provers, but handled well by DPLL(T)-based solvers.

This paper describes a new calculus DPLL(Γ), and an accompanying system, Z3($\mathcal{SP}$), that tightly integrates the strength of saturation based provers in equational reasoning and quantified formulas, with DPLL-based implementations of SMT solvers that are strong in case-analysis and combining theory solvers. The new calculus is very flexible and it can simulate different strategies used in other theorem provers that aim for integrating DPLL and/or theory reasoners as well.

On the technical side, we introduce a key ingredient for this integration; *hypotheses* that track dependencies on case splits when the saturation component performs its deductions. We first lift standard saturation deduction rules to the DPLL(Γ) setting by simply propagating hypotheses (Section 3). It is a somewhat simple, but important, observation that the resulting system is refutationally complete even when ground non-unit clauses are only visible to the DPLL component (but invisible to the inference rules in the saturation component). The lifting becomes less trivial when it comes to using case split literals or any consequence of case splits in deletion and simplification rules. Section 4 presents a lifting that properly tracks case splits into such rules.

On the system side, we discuss the implementation of an instance of DPLL(Γ) in the theorem prover Z3. We believe this is the first report of a really tight integration of DPLL and saturation procedures. Existing systems, to our knowledge, integrate either a black box SMT solver with a saturation prover, [8], or a black box saturation solver with an DPLL core, [9], or don't offer exchanging hypotheses as tightly.

2 Background

We employ basic notions from logic usually assumed in theorem proving. For notation, the symbol $\simeq$ denotes equality; s, u, t are terms; x, y, z are variables; f, g, h, a, b, c are constant or function symbols based on arity; p, q, r are predicate symbols; l is a literal; C and D denote clauses, that is, multi-sets of literals interpreted as disjunctions; $\Box$ is the empty clause; N is a set of clauses; and σ is used for substitutions. A term, literal, or clause is said to be *ground* if it does not contain variables.

We assume that terms are ordered by a simplification ordering $\prec$. It is extended to literals and clauses by a multiset extension. A simplification ordering is well founded and total on ground terms. The most commonly used simplification orderings are instances of the recursive path ordering (RPO) and the Knuth-Bendix ordering (KBO).

An *inference rule* γ is $n + 1$-ary relation on clauses, it is written as:

$$\frac{C_1 \quad \dots \quad C_n}{C}$$

The clauses $C_1, \ldots, C_n$ are called premises, and C the conclusion of the inference. If γ is an inference rule, we denote by $\mathsf{C}(\gamma)$ its conclusion, and $\mathsf{P}(\gamma)$ its premises. An *inference system* Γ is a set of inference rules. We assume that each inference rule has a *main premise* that is "reduced" to the conclusion in the context of the other (*side*) premises.

A *proof* of a clause C from the set of clauses N with respect to an inference system Γ is sequence of clauses $C_1, \ldots, C_n$, where $C_n = C$ and each clause C_i is either an element of N or the conclusion of an inference rule γ of Γ, where the set of premises is a subset of $\{C_1, \ldots, C_{i-1}\}$. A proof of the empty clause $\Box$ is said to be a refutation. An inference system Γ is *refutationally complete* if there is a refutation by Γ from any unsatisfiable set of clauses. A set of clauses N is *saturated* with respect to Γ if the conclusion of any inference by Γ from N is an element of N. Let I be a mapping, called a *model functor*, that assigns to each set of ground clauses N not containing the empty clause an interpretation I_N, called the *candidate model*. If I_N is a model for N, then N is clearly satisfiable. Otherwise, some clause C in N is false in I_N (i.e., C is a counterexample for I_N), then N must contain a *minimal* counterexample with respect to $\prec$. An inference system has the *reduction property for counterexamples*, if for all sets N of clauses and minimal counterexamples C for I_N in N, there is an inference in Γ from N with main premise C, side premises that are true in I_N, and conclusion D that is a smaller counterexample for I_N than C. A clause C is called *redundant* with respect to a set of clauses N if there exists $C_1, \ldots, C_n$ in N such that $C_1, \ldots, C_n \models C$ and $C_i \prec C$ for all $i \in [1, n]$. A set of clauses N is *saturated up to redundancy* with respect to Γ if the conclusion of any inference by Γ with non redundant premises from N is redundant. If Γ is an inference system that satisfies the reduction property for counterexamples, and N a set of clauses saturated up to redundancy, then N is unsatisfiable if, and only if, it contains the empty clause [10].

2.1 Superposition Calculus

The superposition calculus ($\mathcal{SP}$) [11] is a rewriting-based inference system which is refutationally complete for first-order logic with equality. It is based on a simplification order on terms. Figure 1 contains the $\mathcal{SP}$ inference rules. The inference rules restrict generating inferences to positions in maximal terms of maximal literals. More constraints can be imposed, if a clause C contains negative literals, then it is possible to restrict generating inferences to arbitrarily selected negative literals [11].

3 DPLL(Γ)

We will adapt the proof calculus for DPLL(Γ) from an exposition of DPLL(T) as an abstract transition system [1]. DPLL(Γ) is parameterized by a set of inference rules Γ. States of the transition system are of the form $M \parallel F$, where M is a sequence of *decided and implied literals*, and F a set of *hypothetical clauses*. Intuitively, M represents a partial assignment to ground literals and

Equality Resolution

$$\frac{s \not\simeq t \vee C}{\sigma(C)} \quad \textbf{if} \quad \sigma = mgu(s,t), \text{ for all } l \in C,\ \sigma(s \not\simeq t) \not\prec l$$

Equality Factoring

$$\frac{s \simeq t \vee u \simeq v \vee C}{\sigma(t \not\simeq v \vee u \simeq v \vee C)} \quad \textbf{if} \quad \begin{cases} \sigma = mgu(s,u),\ \sigma(s) \not\preceq \sigma(t), \\ \text{for all } l \in (u \simeq v \vee C),\ \sigma(s \simeq t) \not\prec l \end{cases}$$

Superposition Right

$$\frac{s \simeq t \vee C \quad u[s'] \simeq v \vee D}{\sigma(u[t] \simeq v \vee C \vee D)} \quad \textbf{if} \quad \begin{cases} \sigma = mgu(s,s'),\ s' \text{ is not a variable}, \\ \sigma(s) \not\preceq \sigma(t),\ \sigma(u[s']) \not\preceq \sigma(v), \\ \text{for all } l \in C,\ \sigma(s \simeq t) \not\preceq l \\ \text{for all } l \in D,\ \sigma(u[s'] \simeq v) \not\preceq l \end{cases}$$

Superposition Left

$$\frac{s \simeq t \vee C \quad u[s'] \not\simeq v \vee D}{\sigma(u[t] \not\simeq v \vee C \vee D)} \quad \textbf{if} \quad \begin{cases} \sigma = mgu(s,s'),\ s' \text{ is not a variable}, \\ \sigma(s) \not\preceq \sigma(t),\ \sigma(u[s']) \not\preceq \sigma(v), \\ \text{for all } l \in C,\ \sigma(s \simeq t) \not\preceq l \\ \text{for all } l \in D,\ \sigma(u[s'] \not\simeq v) \not\preceq l \end{cases}$$

Fig. 1. Superposition Calculus: Inference Rules

their justifications. During conflict resolution, we also use states of the form $M \,\|\, F \,\|\, C$, where C is a ground clause. A decided literal represents a guess, and an implied literal l_C a literal l that was implied by a clause C. A decided or implied literal in M is said to be an assigned literal. No assigned literal occurs twice in M nor does it occur negated in M. If neither l or $\overline{l}$ occurs in M, then l is said to be undefined in M. We use $lits(M)$ to denote the set of assigned literals. We write $M \models_P C$ to indicate that M propositionally satisfies the clause C. If C is the clause $l_1 \vee \ldots \vee l_n$, then $\neg C$ is the formula $\neg l_1 \wedge \ldots \wedge \neg l_n$. A hypothetical clause is denoted by $H \rhd C$, where H is a set of assigned ground literals (hypotheses) and C is a general clause. The set of hypotheses should be interpreted as a conjunction, and a hypothetical clause $(l_1 \wedge \ldots \wedge l_n) \rhd (l'_1 \vee \ldots l'_m)$ should be interpreted as $\neg l_1 \vee \ldots \vee \neg l_n \vee l'_1 \vee \ldots \vee l'_m$. The basic idea is to allow the inference rules in Γ to use the assigned literals in M as premises, and hypothetical clauses is an artifact to track the dependencies on these assigned literals. We say the conclusions produced by Γ are hypothetical because they may depend on guessed (decided) literals. We use $clauses(F)$ to denote the set $\{C \mid H \rhd C \in F\}$, and $clauses(M \,\|\, F)$ to denote $clauses(F) \cup lits(M)$. We also write C instead of $\emptyset \rhd C$.

The interface with the inference system Γ is realized in the following way: assume γ is an inference rule with n premises, $\{H_1 \rhd C_1, \ldots, H_m \rhd C_m\}$ is a set of hypothetical clauses in F, $\{l_{m+1}, \ldots, l_n\}$ is a set of assigned literals in M, and $\mathsf{H}(\gamma)$ denotes the set $H_1 \cup \ldots \cup H_m \cup \{l_{m+1}, \ldots, l_n\}$, then the inference rule γ is applied to the set of premises $\mathsf{P}(\gamma) = \{C_1, \ldots, C_m, l_{m+1}, \ldots, l_n\}$, and the conclusion $\mathsf{C}(\gamma)$ is added to F as the hypothetical clause $\mathsf{H}(\gamma) \rhd \mathsf{C}(\gamma)$. Note that the hypotheses of the clauses $H_i \rhd C_i$ are hidden from the inference rules in Γ.

Definition 1. *The basic DPLL(Γ) system consists of the following rules:*

Decide

$$M \parallel F \quad \Longrightarrow M\ l \parallel F \quad \textbf{if} \begin{cases} l \textit{ is ground}, \\ l \textit{ or } \bar{l} \textit{ occurs in } F, \\ l \textit{ is undefined in } M. \end{cases}$$

UnitPropagate

$$M \parallel F, H \triangleright (C \vee l) \quad \Longrightarrow M\ l_{H \triangleright (C \vee l)} \parallel F, H \triangleright (C \vee l) \quad \textbf{if} \begin{cases} l \textit{ is ground}, \\ M \models_P \neg C, \\ l \textit{ is undefined in } M. \end{cases}$$

Deduce

$$M \parallel F \quad \Longrightarrow M \parallel F, \mathsf{H}(\gamma) \triangleright \mathsf{C}(\gamma) \quad \textbf{if} \begin{cases} \gamma \in \Gamma, \\ \mathsf{P}(\gamma) \subseteq \textit{clauses}(M \parallel F), \\ \mathsf{C}(\gamma) \notin \textit{clauses}(F) \end{cases}$$

HypothesisElim

$$M \parallel F, (H \wedge l) \triangleright C \quad \Longrightarrow M \parallel F, H \triangleright (C \vee \neg l)$$

Conflict

$$M \parallel F, H \triangleright C \quad \Longrightarrow M \parallel F, H \triangleright C \parallel \neg H \vee C \quad \textbf{if}\ M \models_P \neg C$$

Explain

$$M \parallel F \parallel C \vee \bar{l} \quad \Longrightarrow M \parallel F \parallel \neg H \vee D \vee C \quad \textbf{if}\ l_{H \triangleright (D \vee l)} \in M,$$

Learn

$$M \parallel F \parallel C \quad \Longrightarrow M \parallel F, C \parallel C \quad \textbf{if}\ C \notin \textit{clauses}(F)$$

Backjump

$$M\ l'\ M' \parallel F \parallel C \vee l \Longrightarrow M\ l_{C \vee l} \parallel F' \quad \textbf{if} \begin{cases} M \models_P \neg C, \\ l \textit{ is undefined in } M, \\ F' = \left\{ \begin{array}{l} H \triangleright C \in F \mid \\ \qquad H \cap \textit{lits}(l'\ M') = \emptyset \end{array} \right\} \end{cases}$$

Unsat

$$M \parallel F \parallel \Box \quad \Longrightarrow \textit{unsat}$$

We say a hypothetical clause $H \triangleright C$ is in *conflict* if all literals in C have a complementary assignment. The rule Conflict converts a hypothetical conflict clause $H \triangleright C$ into a regular clause by negating its hypotheses, and puts the DPLL(Γ) system in conflict resolution mode. The Explain rule unfolds literals from conflict clauses that were produced by unit propagation. Any clause derived by Explain can be learned, and added to F, because they are logical consequences of the original set of clauses. The rule Backjump can be used to transition the DPLL(Γ) system back from conflict resolution to search mode, it unassigns at least one decided literal (l' in the rule definition). All hypothetical clauses $H \triangleright C$ which contain hypotheses that will be unassigned by the Backjump rule are deleted.

Figure 2 contains an example that illustrates DPLL(Γ) instantiated with a Γ that contains only the binary resolution and factoring rules. In this example, we annotate each application of the Deduce rule with the premises for the binary resolution inference rule. In this example, the rule HypothesisElim is used to replace the clause $q(a) \triangleright \neg q(y) \vee r(a, y)$ with $\neg q(a) \vee \neg q(y) \vee r(a, y)$ to prevent it from being deleted during backjumping.

Consider the following initial set of clauses:

$$\begin{aligned} F = \{&p(a) \vee p(b), \quad \neg p(b) \vee p(c), \quad \neg p(a) \vee q(a), \quad \neg p(a) \vee q(b), \\ &\neg r(x,b), \quad \neg q(x) \vee \neg q(y) \vee r(x,y), \quad p(x) \vee q(x), \quad \neg q(a) \vee \neg p(b)\} \end{aligned}$$

$$\begin{array}{l}
\quad \| \, F \\
\Longrightarrow \textsf{Decide} \\
\quad p(c) \, \| \, F \\
\Longrightarrow \textsf{Decide} \\
\quad p(c)\ p(a) \, \| \, F \\
\Longrightarrow \textsf{UnitPropagate} \\
\quad p(c)\ p(a)\ q(a)_{\neg p(a) \vee q(a)} \, \| \, F \\
\Longrightarrow \textsf{Deduce} \ \ \textit{resolution } q(a) \textit{ with } \neg q(x) \vee \neg q(y) \vee r(x,y) \\
\quad p(c)\ p(a)\ q(a)_{\neg p(a) \vee q(a)} \, \| \, F,\ q(a) \triangleright \neg q(y) \vee r(a,y) \\
\Longrightarrow \textsf{UnitPropagate} \\
\quad p(c)\ p(a)\ q(a)_{\neg p(a) \vee q(a)}\ q(b)_{\neg p(a) \vee q(b)} \, \| \, F,\ q(a) \triangleright \neg q(y) \vee r(a,y) \\
\Longrightarrow \textsf{Deduce} \ \ \textit{resolution } q(b) \textit{ with } \neg q(y) \vee r(a,y) \\
\quad p(c)\ p(a)\ q(a)_{\neg p(a) \vee q(a)}\ q(b)_{\neg p(a) \vee q(b)} \, \| \, F,\ q(a) \triangleright \neg q(y) \vee r(a,y),\ q(a) \wedge q(b) \triangleright r(a,b) \\
\Longrightarrow \textsf{HypothesisElim} \\
\quad p(c)\ p(a)\ q(a)_{\neg p(a) \vee q(a)}\ q(b)_{\neg p(a) \vee q(b)} \, \| \, \underbrace{F,\ \neg q(a) \vee \neg q(y) \vee r(a,y),\ q(a) \wedge q(b) \triangleright r(a,b)}_{F'} \\
\Longrightarrow \textsf{Deduce} \ \ \textit{resolution } \neg r(x,b) \textit{ with } r(a,b) \\
\quad p(c)\ p(a)\ q(a)_{\neg p(a) \vee q(a)}\ q(b)_{\neg p(a) \vee q(b)} \, \| \, F',\ q(a) \wedge q(b) \triangleright \square \\
\Longrightarrow \textsf{Conflict} \\
\quad p(c)\ p(a)\ q(a)_{\neg p(a) \vee q(a)}\ q(b)_{\neg p(a) \vee q(b)} \, \| \, F',\ q(a) \wedge q(b) \triangleright \square \, \| \, \neg q(a) \vee \neg q(b) \\
\Longrightarrow \textsf{Explain} \\
\quad p(c)\ p(a)\ q(a)_{\neg p(a) \vee q(a)}\ q(b)_{\neg p(a) \vee q(b)} \, \| \, F',\ q(a) \wedge q(b) \triangleright \square \, \| \, \neg p(a) \vee \neg q(b) \\
\Longrightarrow \textsf{Explain} \\
\quad p(c)\ p(a)\ q(a)_{\neg p(a) \vee q(a)}\ q(b)_{\neg p(a) \vee q(b)} \, \| \, F',\ q(a) \wedge q(b) \triangleright \square \, \| \, \neg p(a) \\
\Longrightarrow \textsf{Backjump} \\
\quad \neg p(a)_{\neg p(a)} \, \| \, F,\ \neg q(a) \vee \neg q(y) \vee r(a,y) \\
\Longrightarrow \textsf{UnitPropagate} \\
\quad \neg p(a)_{\neg p(a)}\ p(b)_{p(a) \vee p(b)} \, \| \, F,\ \neg q(a) \vee \neg q(y) \vee r(a,y) \\
\Longrightarrow \textsf{Deduce} \ \ \textit{resolution } \neg p(a) \textit{ with } p(x) \vee q(x) \\
\quad \neg p(a)_{\neg p(a)}\ p(b)_{p(a) \vee p(b)} \, \| \, \underbrace{F,\ \neg q(a) \vee \neg q(y) \vee r(a,y),\ \neg p(a) \triangleright q(a)}_{F''} \\
\Longrightarrow \textsf{UnitPropagate} \\
\quad \neg p(a)_{\neg p(a)}\ p(b)_{p(a) \vee p(b)}\ q(a)_{\neg p(a) \triangleright q(a)} \, \| \, F'' \\
\Longrightarrow \textsf{Conflict} \\
\quad \neg p(a)_{\neg p(a)}\ p(b)_{p(a) \vee p(b)}\ q(a)_{\neg p(a) \triangleright q(a)} \, \| \, F'' \, \| \, \neg q(a) \vee \neg p(b) \\
\Longrightarrow \textsf{Explain} \\
\quad \neg p(a)_{\neg p(a)}\ p(b)_{p(a) \vee p(b)}\ q(a)_{\neg p(a) \triangleright q(a)} \, \| \, F'' \, \| \, p(a) \vee \neg p(b) \\
\Longrightarrow \textsf{Explain} \\
\quad \neg p(a)_{\neg p(a)}\ p(b)_{p(a) \vee p(b)}\ q(a)_{\neg p(a) \triangleright q(a)} \, \| \, F'' \, \| \, p(a) \\
\Longrightarrow \textsf{Explain} \\
\quad \neg p(a)_{\neg p(a)}\ p(b)_{p(a) \vee p(b)}\ q(a)_{\neg p(a) \triangleright q(a)} \, \| \, F'' \, \| \, \square \\
\Longrightarrow \textsf{Unsat} \\
\quad \textit{unsat}
\end{array}$$

Fig. 2. DPLL(Γ): example

3.1 Soundness and Completeness

We assume that the set of inference rules Γ is sound and refutationally complete. Then, it is an easy observation that all transition rules in DPLL(Γ) preserve satisfiability. In particular, all clauses added by conflict resolution are consequences of the original set of clauses F.

Theorem 1 (Soundness). *DPLL(Γ) is sound.*

From any unsatisfiable set of clauses, a fair application of the transition rules in DPLL(Γ) will eventually generate the *unsat* state. This is a direct consequence of the refutational completeness of Γ. That is, Deduce alone can be used to derive the empty clause without using any hypothesis.

Theorem 2 (Completeness). *DPLL(Γ) is refutationally complete.*

Unrestricted use of the Deduce rule described in Defintion 1 is not very effective, because it allows the inference rules in Γ to use arbitrary ground clauses as premises. We here introduce a refinement of Deduce, called Deduce$^\sharp$, which applies on fewer cases than Deduce, but still maintains refutational completeness. In Deduce$^\sharp$, the set of premises for inference rules in Γ are restricted to non-ground clauses and ground unit clauses. That is, $\mathsf{P}(\gamma) \subseteq \mathit{premises}(M \parallel F)$, where we define $\mathit{nug}(N)$ to be the subset of non unit ground clauses of a set of clauses N, and $\mathit{premises}(M \parallel F) = (\mathit{clauses}(F) \setminus \mathit{nug}(\mathit{clauses}(F))) \cup \mathit{lits}(M)$. The refined rule is then:

Deduce$^\sharp$

$$M \parallel F \implies M \parallel F, \mathsf{H}(\gamma) \triangleright \mathsf{C}(\gamma) \qquad \textbf{if} \begin{cases} \gamma \in \Gamma, \quad \mathsf{P}(\gamma) \subseteq \mathit{premises}(M \parallel F), \\ \mathsf{C}(\gamma) \notin \mathit{clauses}(F) \end{cases}$$

The idea is to use DPLL to handle all case-analysis due to ground clauses. The refined system is called DPLL(Γ)$^\sharp$. A state $M \parallel F$ of DPLL(Γ)$^\sharp$ is said to be *saturated* if any ground literal in F is assigned in M, there is no ground clause C in $\mathit{clauses}(F)$ such that $M \models_P \neg C$, and if the conclusion of any inference by Γ from $\mathit{premises}(M \parallel F)$ is an element of $\mathit{clauses}(F)$.

Theorem 3. *If $M \parallel F$ is a saturated state of DPLL(Γ)$^\sharp$ for an initial set of clauses N and Γ has the reduction property for counterexamples, then N is satisfiable.*

Proof. Since all transitions preserve satisfiability, we just need to show that $\mathit{clauses}(F) \cup \mathit{lits}(M)$ is satisfiable. The set of clauses $\mathit{clauses}(F) \cup \mathit{lits}(M)$ is not saturated with respect to Γ because clauses in $\mathit{nug}(\mathit{clauses}(F))$ were not used as premises for its inference rules, but the set is saturated up to redundancy. Any clause C in $\mathit{nug}(\mathit{clauses}(F))$ is redundant because there is a literal l of C that is in $\mathit{lits}(M)$, and clearly $l \models C$ and $l \prec C$. Since Γ has the reduction property for counterexamples, the set $\mathit{clauses}(F) \cup \mathit{lits}(M)$ is satisfiable.

We assign a *proof depth* to any clause in *clauses*(F) and literal in *lits*(M). Intuitively, the proof depth of a clause C indicates the depth of the derivation needed to produce C. More precisely, all clauses in the original set of clauses have proof depth 0. If a clause C is produced using the Deduce rule, and n is the maximum proof depth of the premises, then the proof depth of C is $n+1$. The proof depth of a literal l_C in M is equal to the proof depth of C. If l is a decided literal, and n is the minimum proof depth of the clauses in F that contain l, then the proof depth of l is n. We say DPLL$(\Gamma)^\sharp$ is k-bounded if Deduce$^\sharp$ is restricted to premises with proof depth $< k$. Note that, the number of ground literals that can be produced in a k-bounded run of DPLL$(\Gamma)^\sharp$ is finite.

Theorem 4. *A k-bounded DPLL$(\Gamma)^\sharp$ always terminates.*

The theorem above is also true for DPLL(Γ). In another variation of DPLL(Γ), the restriction on Deduce$^\sharp$ used in DPLL$(\Gamma)^\sharp$ is disabled after k steps. This variation is also refutationally complete, since all transition rules preserve satisfiability and Γ is refutationally complete.

3.2 Additional Rules

SMT solvers implement efficient theory reasoning for conjunctions of ground literals. One of the main motivations of DPLL(Γ) is to use these efficient theory solvers in conjunction with arbitrary inference systems. Theory reasoning is incorporated in DPLL(Γ) using transition rules similar to the one described in [1]. Unfortunately, the theorems presented in the previous section do not hold in general when theory reasoning is incorporated. We use $F \models_T G$ to denote the fact that F *entails* G in theory T.

T-Propagate

$$M \parallel F \quad \Longrightarrow M\ l_{(\neg l_1 \vee \ldots \vee \neg l_n \vee l)} \parallel F \quad \textbf{if} \begin{cases} l \text{ is ground and occurs in } F, \\ l \text{ is undefined in } M, \\ l_1, \ldots, l_n \in \mathit{lits}(M) \\ l_1, \ldots, l_n \models_T l, \end{cases}$$

T-Conflict

$$M \parallel F \quad \Longrightarrow M \parallel F \parallel \neg l_1 \vee \ldots \vee \neg l_n \quad \textbf{if} \begin{cases} l_1, \ldots, l_n \in \mathit{lits}(M), \\ l_1, \ldots, l_n \models_T \mathit{false} \end{cases}$$

4 Contraction Inferences

Most modern saturation theorem provers spend a considerable amount of time simplifying and eliminating redundant clauses. Most of them contain *simplifying* and *deleting inferences*. We say these are *contraction inferences*. Although these inferences are not necessary for completeness, they are very important in practice. This section discusses how contraction inferences can be incorporated into DPLL(Γ). We distinguish between contraction inferences taking 1 premise, such as deletion of duplicate and resolved literals, tautology deletion, and destructive equality resolution, and rules that take additional clauses besides the one being simplified or eliminated (e.g., subsumption). Assume the contraction rules in a saturation system are described as $\gamma_{d_1}, \gamma_{s_1}, \gamma_d$, and γ_s. Common to these rules is they take a set of clauses F, C and either *delete* C, producing F or *simplify* C to C', producing F, C'. We call C the *main premise*, and other premises are called *side premises*. Thus, we will here be lifting to DPLL(Γ) rules of the form:

$$\frac{F, C}{F}\,\gamma_{d_1}(C) \qquad\qquad \frac{F, C, C_2, \ldots, C_n}{F, C_2, \ldots, C_n}\,\gamma_d(C, C_2, \ldots, C_n),\ n \geq 2$$

$$\frac{F, C}{F, C'}\,\gamma_{s_1}(C, C') \qquad\qquad \frac{F, C, C_2, \ldots, C_n}{F, C', C_2, \ldots, C_n}\,\gamma_s(C, C_2, \ldots, C_n, C'),\ n \geq 2$$

Contraction rules. Any contraction inference γ_{d_1} or γ_{s_1} that contains only one premise can be easily incorporated into DPLL(Γ). Given a hypothetical clause $H \triangleright C$, the contraction rules are just applied to C. For contraction rules with more than one premise (e.g., subsumption), special treatment is needed. For example, consider the following state:

$$p(a) \parallel p(a) \triangleright p(b),\quad p(b) \vee p(c),\quad p(a) \vee p(b)$$

In this state, the clause $p(b)$ subsumes the clause $p(b) \vee p(c)$, but it is not safe to delete $p(b) \vee p(c)$ because the subsumer has a hypothesis. That is, after backjumping decision literal $p(a)$, $p(a) \triangleright p(b)$ will be deleted and $p(b) \vee p(c)$ will not be subsumed anymore. A naïve solution consists in using the HypothesisElim to transform hypothetical clauses $H \triangleright C$ into regular clauses $\neg H \vee C$. This solution is not satisfactory because important contraction rules, such as demodulation, have premises that must be unit clauses. Moreover, HypothesisElim also has the unfortunate side-effect of eliminating relationships between different hypotheses. For example, in the following state:

$$p(a)\ p(b)_{\neg p(a) \vee p(b)} \parallel p(a) \triangleright p(c),\quad p(b) \triangleright p(c) \vee p(d)$$

it is safe to use the clause $p(c)$ to subsume $p(c) \vee p(d)$ because $p(a) \triangleright p(c)$ and $p(b) \triangleright p(c) \vee p(d)$ will be deleted at the same time during backjumping. To formalize this approach, we assign a *scope level* to any assigned literal in M. The scope level of a literal l (*level*(l)) in $M\ l\ M'$ equals to the number of decision literals in $M\ l$. For example, in the following state:

$$p(a)_{p(a)}\ p(b)\ p(c)_{\neg p(b) \vee p(c)}\ p(d) \parallel \ldots$$

The scope levels of $p(a)$, $p(b)$, $p(c)$ and $p(d)$ are 0, 1, 1 and 2, respectively. The level of a set of literals is the supremum level, thus for a set H, $\mathit{level}(H) = \mathit{max}\{\mathit{level}(l) \mid l \in H\}$. Clearly, if literal l occurs after literal l' in M, then $\mathit{level}(l) \geq \mathit{level}(l')$. In the Backjump rule we go from a state $M\ l'\ M' \parallel F \parallel C \vee l$ to a state $M\ l_{C \vee l} \parallel F'$, and we say the scope level $\mathit{level}(l')$ was *backjumped*.

We now have the sufficient background to formulate how deletion rules with multiple premises can be lifted in DPLL(Γ). Thus, we seek to lift γ_d to a main clause of the form $H \triangleright C$, and side clauses $H_2 \triangleright C_2, \ldots, H_m \triangleright C_m, l_{m+1}, \ldots, l_n$ taken from F and $lits(M)$. The hypothesis of the main clause is H and the hypotheses used in the side clauses are $H_2 \cup \ldots \cup H_m \cup \{l_{m+1}, \ldots, l_n\}$ (called H' for short). So assume the premise for deletion holds, that is $\gamma_d(C, C_2, \ldots, C_m, l_{m+1}, \ldots, l_n)$. If $level(H) \geq level(H')$, we claim it is safe to delete the main clause $H \triangleright C$. This is so as backjumping will never delete side premises before removing the main premise. Thus, it is not possible to backjump to a state where one of the side premises was deleted from F or M, but a main premise from H is still *alive*: a deleted clause would not be redundant in the new state. In contrast, if $level(H) < level(H')$, then it is only safe to *disable* the clause $H \triangleright C$ until $level(H')$ is backjumped.

A *disabled* clause is not deleted, but is not used as a premise for any inference rule until it is re-enabled. To realize this new refinement, we maintain one array of disabled clauses for every scope level. Clauses that are disabled are not deleted, but moved to the array at the scope level of H'. Clauses are moved from the array of disabled clauses back to the set of main clauses F when backjumping pops scope levels. We annotate disabled clauses as $[H \triangleright C]_k$, where k is the level at which the clause can be re-enabled.

Contraction rules summary. We can now summarize how the contraction rules lift to DPLL(Γ). In the contraction rules below, assume $H_2 \triangleright C_2, \ldots, H_m \triangleright C_m \in F$, $l_{m+1}, \ldots, l_n \in lits(M)$, and $H' = H_2 \cup \ldots \cup H_m \cup \{l_{m+1}, \ldots, l_n\}$.

Delete

$$M \parallel F, H \triangleright C \quad \Longrightarrow M \parallel F \qquad \textbf{if} \begin{cases} \gamma_d(C, C_2, \ldots, l_n), \ n \geq 2 \\ level(H) \geq level(H') \end{cases}$$

Disable

$$M \parallel F, H \triangleright C \quad \Longrightarrow M \parallel F, [H \triangleright C]_{level(H')} \quad \textbf{if} \begin{cases} \gamma_d(C, C_2, \ldots, l_n), \ n \geq 2 \\ level(H) < level(H') \end{cases}$$

Simplify

$$M \parallel F, H \triangleright C \quad \Longrightarrow M \parallel F, (H \cup H') \triangleright C' \quad \textbf{if} \begin{cases} \gamma_s(C, C_2, \ldots, l_n, C'), \ n \geq 2 \\ level(H) \geq level(H') \end{cases}$$

Simplify-disable

$$M \parallel F, H \triangleright C \quad \Longrightarrow M \parallel F, [H \triangleright C]_{level(H')}, (H \cup H') \triangleright C' \quad \textbf{if} \begin{cases} \gamma_s(C, C_2, \ldots, l_n, C'), \ n \geq 2 \\ level(H) < level(H') \end{cases}$$

We also use Delete_1 and $\mathsf{Simplify}_1$ as special cases (without side conditions) of the general rules for γ_{d_1} and γ_{s_1}.

5 System Architecture

We implemented DPLL(Γ) (and DPLL(Γ)$^\sharp$) in the Z3 theorem prover [6] by instantiating the calculus with the $\mathcal{SP}$ inference rules. Z3 is a state of the art SMT solver which previously used E-matching exclusively for handling quantified formulas. It integrates a modern DPLL-based SAT solver, a core theory solver that handles ground equalities over uninterpreted functions, and *satellite solvers* (for arithmetic, bit-vectors, arrays, etc). The new system is called Z3($\mathcal{SP}$). It uses perfectly shared expressions as its main data structure. This data structure is implemented in the standard way using hash-consing. These expressions are used by the DPLL(T) and $\mathcal{SP}$ engines.

DPLL(T) engine. The rules Decide, UnitPropagate, Conflict, Explain, Learn, Backjump, and Unsat are realized by the DPLL-based SAT solver in Z3($\mathcal{SP}$)'s core. During exploration of a particular branch of the search tree, several of the assigned literals are not relevant. Relevancy propagation [12] keeps track of which truth assignments in M are essential for determining satisfiability of a formula. In our implementation, we use a small refinement where only literals that are marked as relevant have their truth assignment propagated to theory solvers and are made available as premises to the Deduce rule. The rules T-Propagate and T-Conflict are implemented by the congruence closure core and satellite solvers. The congruence closure core processes the assigned literals in M. Atoms range over equalities and theory specific atomic formulas, such as arithmetical inequalities. Equalities asserted in M are propagated by the congruence closure core using a data structure that we will call an E-graph following [2]. Each expression is associated with an E-node that contains the extra fields used to implement the Union-find algorithm, congruence closure, and track references to theory specific data structures. When two expressions are merged, the merge is propagated as an equality to the relevant theory solvers. The core also propagates the effects of the theory solvers, such as inferred equalities that are produced and atoms assigned to true or false. The theory solvers may also produce new ground clauses in the case of non-convex theories. These ground clauses are propagated to F.

$\mathcal{SP}$ engine. The rules Deduce, Deduce$^\sharp$, Delete, Disable and Simplify are implemented by the new $\mathcal{SP}$ engine. It contains a superset of the $\mathcal{SP}$ rules described in Figure 1, that includes contraction rules such as: forward/backward rewriting, forward/backward subsumption, tautology deletion, destructive equality resolution and equality subsumption. Z3($\mathcal{SP}$) implements Knuth-Bendix ordering (KBO) and lexicographical path ordering (LPO). Substitution trees [13] is the main indexing data structure used in the $\mathcal{SP}$ engine. It is used to implement most of the inference rules: forward/backward rewriting, superposition right/left, binary resolution, and unit-forward subsumption. Feature vector indexing [14] is used to implement non-unit forward and backward subsumption. Several inference rules assume that premises have no variables in common. For performance reasons, we do not explicitly rename variables, but use *expression offsets* like the Darwin theorem prover [15]. Like most saturation theorem provers, we store in each clause the set of *parent clauses* (premises) used to infer it. Thus, the set of hypotheses of an inferred clause is only computed during conflict resolution by following the pointers to *parent clauses*. This is an effective optimization because most of the inferred clauses are never used during conflict resolution.

Rewriting with ground equations. Due to the nature of our benchmarks and DPLL(Γ), most of the equations used for forward rewriting are ground. It is wasteful to use substitution trees as an indexing data structure in this case.

When rewriting a term t, for every subterm s of t we need to check if there is an equation $s' \simeq u$ such that s is an instance of s'. With substitution trees, this operation may consume $\mathcal{O}(n)$ time where n is the size of s. If the equations are ground, we can store the equations as a mapping G, where $s \mapsto u \in G$ if $s \simeq u$ is ground and $u \prec s$. We can check in constant time if an expression s is a key in the mapping G, because we use perfectly shared terms.

Ground equations. Ground equations are processed by the congruence closure core and $\mathcal{SP}$ engine. To avoid duplication of work, we convert the E-graph into a canonical set of ground rewrite rules using the algorithm described in [16]. This algorithm is attractive in our context because its first step consists of executing a congruence closure algorithm.

E-matching for theories. Although the new system is refutationally complete for first-order logic with equality, it fails to prove formulas containing theory reasoning that could be proved using E-matching. For example, consider the following simple unsatisfiable set of formulas:

$$\neg(f(a) > 2), \quad f(x) > 5$$

DPLL(Γ) fails to prove the unsatisfiability of this set because 2 does not unify with 5. On the other hand, the E-matching engine selects $f(x)$ as a pattern (trigger), instantiates the quantified formula $f(x) > 5$ with the substitution $[x \mapsto a]$, and the arithmetic solver detects the inconsistency. Thus, we still use the E-matching engine in Z3($\mathcal{SP}$). E-matching can be described as:

E-matching

$$M \parallel F, H \triangleright C[t] \quad \Longrightarrow \quad M \parallel F, H \triangleright C[t], \sigma(H \triangleright C[t]) \;\textbf{if} \begin{cases} E \models \sigma(t) \simeq s, \text{ where} \\ s \text{ is a ground term in } M, \\ E = \{s_1 \simeq r_1, ..\} \subseteq M \end{cases}$$

Definitions. In software verification, more precisely in Spec♯ [17] and VCC (Verified C Complier), the verification problems contain a substantial number of definitions of the form: $p(\bar{x}) \Leftrightarrow C[\bar{x}]$. Moreover, most of these definitions are irrelevant for a given problem. Assuming $C[\bar{x}]$ is a clause $l_1[\bar{x}] \vee \ldots \vee l_n[\bar{x}]$, the definition is translated into the following set of clauses $Ds = \{\neg p(\bar{x}) \vee l_1[\bar{x}] \vee \ldots \vee l_n[\bar{x}],\ \ p(\bar{x}) \vee \neg l_1[\bar{x}], \ldots, p(\bar{x}) \vee \neg l_n[\bar{x}]\}$. We avoid the overhead of processing irrelevant definitions by using a term ordering that makes the literal containing $p(\bar{x})$ maximal in the clauses Ds.

Search heuristic. A good search heuristic is crucial for a theorem prover. Our default heuristic consists in eagerly applying UnitPropagate, T-Propagate, Conflict and T-Conflict. Before each Decide, we apply k times E-matching and Deduce. The value k is small if there are unassigned ground clauses in F. The E-matching rule is mainly applied to quantified clauses that contains theory symbols. The rule Deduce is implemented using a variant of the *given-clause* algorithm, where the $\mathcal{SP}$ engine state has two sets of clauses: P processed and U unprocessed. Similarly to E [18], the clause selection strategy can use an arbitrary number of priority queues.

Candidate models. Software verification tools such as Spec$\sharp$ and VCC expect the theorem prover to provide a model for satisfiable formulas. Realistically, the theorem prover produces only candidate (potential) models. In Z3($\mathcal{SP}$), M is used to produce the candidate model, and a notion of *k-saturation* is used. We say a formula is *k-maybe-sat*, if all ground literals in M are assigned, and Deduce cannot produce any new non redundant clause with proof depth $< k$.

5.1 Evaluation

Benchmarks. We considered three sets of benchmarks: NASA [7], ESC/Java, and Spec$\sharp$ [17]. The NASA benchmarks are easy even for SMT solvers based on E-matching. In the 2007 SMT competition[1] the selected NASA benchmarks were all solved in less than one sec by all competitors. The Simplify theorem prover fails on some of these benchmarks because they do not contain hints (triggers), and Simplify uses a very conservative trigger selection heuristic. The ESC/Java and Spec$\sharp$ benchmarks are similar in nature, and are substantially more challenging than the NASA benchmarks[2]. These benchmarks contain hints for provers based on E-matching. These hints can be used also by the $\mathcal{SP}$ engine to guide the literal selection strategy. If the trigger (hint) is contained in a negative literal l, we simply select l for generating inferences. Surprisingly, a substantial subset of the axioms can be handled using this approach. When the trigger is contained in a positive literal, Z3($\mathcal{SP}$) tries to make it maximal. This heuristic is particularly effective with the *frame axioms* automatically generated by Spec$\sharp$:

$$C[a, a', x, y] \vee read(a, x, y) = read(a', x, y)$$

where $C[a, a', x, y]$ contains 10 literals, and specifies which locations of the heap were not modified. In Spec$\sharp$, the heap is represented as a bidimensional array. The Spec$\sharp$ benchmarks contain the transitivity and monotonicity axioms:

$$\neg p(x, y) \vee \neg p(y, z) \vee p(x, z), \quad \neg p(x, y) \vee p(f(x), f(y))$$

These axioms are used to model Spec$\sharp$'s type system. Each benchmark contains several ground literals of the form $p(s, t)$. The transitivity axiom can be easily "tamed" by selecting one of its negative literals. Unfortunately, this simple approach does not work with the monotonicity axiom. An infinite sequence of irrelevant clauses is produced by the $\mathcal{SP}$ engine. In contrast, unsatisfiable instances containing these axioms can be easily handled by solvers based on E-matching. At this point, we do not have a satisfactory solution for this problem besides giving preference to clauses with small proof depth.

[1] http://www.smtcomp.org

[2] All benchmarks are available at: http://www.smtlib.org.
Z3 runtime logs for NASA's and Spec$\sharp$'s benchmarks can be found at: http://research.microsoft.com/users/leonardo/IJCAR2008.

Inconsistent axioms. A recurrent problem that was faced by Z3 users was sets of axioms that were unintentionally inconsistent. An E-matching based solver usually fails to detect the inconsistency. Even worse, in some cases, small modifications in the formula allowed the prover to transition from failure to success and vice-versa. The following unsatisfiable set of formulas were extracted from one of these problematic benchmarks:

$$a \neq b, \quad \neg p(x,c) \vee p(x,b), \quad \neg p(x,y) \vee x = y, \quad p(x,c)$$

The inconsistency is not detected by E-matching because there is no ground literal using the predicate p. Now, assume the formula also contains the clause $q(a) \vee p(a,b)$. If the prover decides to satisfy this clause by assigning $q(a)$, then $p(a,b)$ is ignored by the E-matching engine and the prover fails. On the other hand, if $p(a,b)$ is selected by the prover, then it is used for instantiation and the proof succeeds. In contrast, Z3($\mathcal{SP}$) can easily detect the inconsistency.

6 Related Work

HaRVey-FOL [9] combines Boolean solving with the equational theorem prover E. Its main application is software verification. The Boolean solver is used to handle the control flow of software systems, E allows haRVey to provide support for a wide range of theories formalizing data-structures. The integration is loose in this case, for example, a non-unit ground clause produced by E is not processed by the Boolean solver. SPASS+T [8] integrates an arbitrary SMT solver with the SPASS [19] saturation theorem prover. The SMT solver is used to handle arithmetic and free functions. The integration is also loose, SPASS uses its deduction rules to produce new formulas as usual, and the ground formulas are propagated to the SMT solver. The ground formulas are processed by both systems. Moreover, the clause splitting available in SPASS cannot be used because SPASS+T has no control over the (black box) SMT solver backtracking search. SMELS [20], to our best understanding, is a method for incorporating theory reasoning in efficient DPLL + congruence closure solvers. The ground clauses are sent to the DPLL solver. The congruence closure algorithm calculates all reduced (dis)equalities. A Superposition procedure then performs an inference rule, which is called Justified Superposition, between these (dis)equalities and the nonground clauses. The resulting ground clauses are provided to the DPLL engine.

LASCA [21] is a method for integrating linear rational arithmetic into superposition calculus for first-order logic. One of the main results is completeness of the calculus under some finiteness assumptions. Like regular equational theorem provers, the calculus does not have support for performing efficient case-analysis like a DPLL-based calculus.

SPASS [19] theorem prover supports *splitting*. The idea is to combine the β-rule found in tableau calculi with a saturation based prover. Their formalization relies on the use of labels [22], the main idea is to label clauses with the split levels they depend on. In contrast, the main advantage of the approach used in DPLL(Γ) is better integration with the conflict resolution techniques (i.e., backjumping and lemma learning) used in DPLL-based SAT solvers. In [23], an alternative approach to case analysis for saturation theorem provers is described. The main idea is to introduce new special propositional symbols. This approach has two main disadvantages with respect to DPLL(Γ): the generated clauses can not be directly used for reductions, and efficient conflict resolution techniques used in DPLL-based solvers cannot be directly applied.

7 Conclusion

We have introduced a calculus that tightly integrates inference rules used in saturation based provers with the DPLL(T) calculus. The combination is refutationally complete for first-order logic. The calculus is particulary attractive for software verification because all non-unit ground clauses can be delegated to DPLL(T). We believe this work also provides a better and more flexible infrastructure for implementing the rewrite-based approach for decision procedures [24]. The DPLL(Γ) calculus was implemented in the Z3($\mathcal{SP}$) theorem prover, and the main design decisions were discussed.

References

1. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). J. ACM 53, 937–977 (2006)
2. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM 52, 365–473 (2005)
3. Ge, Y., Barrett, C., Tinelli, C.: Solving quantified verification conditions using satisfiability modulo theories. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603. Springer, Heidelberg (2007)
4. Moskal, M., Lopuszanski, J., Kiniry, J.R.: E-matching for fun and profit. In: Satisfiability Modulo Theories Workshop (2007)
5. 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)
6. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: TACAS 2008 (2008)
7. Denney, E., Fischer, B., Schumann, J.: Using automated theorem provers to certify auto-generated aerospace software. In: Basin, D., Rusinowitch, M. (eds.) IJCAR 2004. LNCS (LNAI), vol. 3097. Springer, Heidelberg (2004)
8. Prevosto, V., Waldmann, U.: SPASS+T. In: ESCoR Workshop (2006)
9. Deharbe, D., Ranise, S.: Satisfiability solving for software verification. International Journal on Software Tools Technology Transfer (to appear, 2008)
10. Bachmair, L., Ganzinger, H.: Resolution theorem proving. In: Handbook of Automated Reasoning, pp. 19–99. MIT Press, Cambridge (2001)
11. Nieuwenhuis, R., Rubio, A.: Paramodulation-based theorem proving. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning. MIT Press, Cambridge (2001)

12. de Moura, L., Bjørner, N.: Relevancy Propagation. Technical Report MSR-TR-2007-140, Microsoft Research (2007)
13. Graf, P.: Substitution tree indexing. In: Hsiang, J. (ed.) RTA 1995. LNCS, vol. 914. Springer, Heidelberg (1995)
14. Schulz, S.: Simple and Efficient Clause Subsumption with Feature Vector Indexing. In: ESFOR Workshop (2004)
15. Baumgartner, P., Fuchs, A., Tinelli, C.: Darwin: A theorem prover for the model evolution calculus. In: ESFOR Workshop (2004)
16. Gallier, J., Narendran, P., Plaisted, D., Raatz, S., Snyder, W.: An algorithm for finding canonical sets of ground rewrite rules in polynomial time. J. ACM 40 (1993)
17. 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, pp. 49–69. Springer, Heidelberg (2005)
18. Schulz, S.: E - a brainiac theorem prover. AI Commun 15, 111–126 (2002)
19. Weidenbach, C., Brahm, U., Hillenbrand, T., Keen, E., Theobalt, C., Topic, D.: SPASS version 2.0. In: Voronkov, A. (ed.) CADE 2002. LNCS (LNAI), vol. 2392, Springer, Heidelberg (2002)
20. Lynch, C.: SMELS: Satisfiability Modulo Equality with Lazy Superposition. The Dagsthul Seminar on Deduction and Decision Procedures (2007)
21. 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)
22. Fietzke, A.: Labelled splitting. Master's thesis, Saarland University (2007)
23. Riazanov, A., Voronkov, A.: Splitting without backtracking. In: IJCAI (2001)
24. Armando, A., Bonacina, M.P., Ranise, S., Schulz, S.: New results on rewrite-based satisfiability procedures. ACM Transactions on Computational Logic (to appear)

THF0 – The Core of the TPTP Language for Higher-Order Logic

Christoph Benzmüller[1,⋆], Florian Rabe[2], and Geoff Sutcliffe[3]

[1] Saarland University, Germany
[2] Jacobs University Bremen, Germany
[3] University of Miami, USA

Abstract. One of the keys to the success of the Thousands of Problems for Theorem Provers (TPTP) problem library and related infrastructure is the consistent use of the TPTP language. This paper introduces the core of the TPTP language for higher-order logic – THF0, based on Church's simple type theory. THF0 is a syntactically conservative extension of the untyped first-order TPTP language.

1 Introduction

There is a well established infrastructure that supports research, development, and deployment of first-order Automated Theorem Proving (ATP) systems, stemming from the Thousands of Problems for Theorem Provers (TPTP) problem library [29]. This infrastructure includes the problem library itself, the TPTP language [27], the SZS ontologies [30], the Thousands of Solutions from Theorem Provers (TSTP) solution library, various tools associated with the libraries [26], and the CADE ATP System Competition (CASC) [28]. This infrastructure has been central to the progress that has been made in the development of high performance first-order ATP systems.

One of the keys to the success of the TPTP and related infrastructure is the consistent use of the TPTP language. Until now the TPTP language has been defined for only untyped first-order logic (first-order form (FOF) and clause normal form (CNF)). This paper introduces the core of the TPTP language for classical higher-order logic – THF0, based on Church's simple type theory. THF0 is the *core* language in that it provides only the commonly used and accepted aspects of a higher-order logic language. The full THF language includes the THFF extension that allows the use of first-order style prefix syntax, and the THF1 and THF2 extensions that provide successively richer constructs in higher-order logic.[1]

⋆ This work was supported by EPSRC grant EP/D070511/1 (LEO-II).

[1] The THFF, THF1, and THF2 extensions have already been completed. The inital release of only THF0 allows users to adopt the language without being swamped by the richness of the full THF language. The full THF language definition is available from the TPTP web site, www.tptp.org

As with the first-order TPTP language, a common adoption of the THF language will enable convenient communication of higher-order data between different systems and researchers. THF0 is the starting point for building higher-order analogs of the existing first-order infrastructure components, including a higher-order extension to the TPTP problem library, evaluation of higher-order ATP systems in CASC, TPTP tools to process THF0 problems, and a higher-order TSTP proof representation format. An interesting capability that has become possible with the THF language is easy comparison of higher-order and first-order versions of problems, and evaluation of the relative benefits of the different encodings with respect to ATP systems for the logics.

2 Preliminaries

2.1 The TPTP Language

The TPTP language is a human-readable, easily machine-parsable, flexible and extensible language suitable for writing both ATP problems and solutions. The BNF of the THF0 language, which defines common TPTP language constructs and the THF0 specific constructs, is given in Appendix A.

The top level building blocks of the TPTP language are *annotated formulae*, *include directives*, and *comments*. An annotated formula has the form:

language`(`*name*`,` *role*`,` *formula*`,` `[`*source*`,` `[`*useful_info*`]])`.

An example annotated first-order formula, supplied from a file, is:

```
fof(formula_27,axiom,
    ! [X,Y] :
      ( subclass(X,Y)
    <=> ! [U] :
          ( member(U,X)
         => member(U,Y) )),
    file('SET005+0.ax',subclass_defn),
    [description('Definition of subclass'), relevance(0.9)]).
```

The *language*s supported are first-order form (`fof`), clause normal form (`cnf`), and now typed higher-order form (`thf`). The *role*, e.g., `axiom`, `lemma`, `conjecture`, defines the use of the formula in an ATP system - see the BNF for the list of recognized roles. The details of the *formula* notation for connectives and quantifiers can be seen in the BNF. The forms of identifiers for uninterpreted functions, predicates, and variables follow Prolog conventions, i.e., functions and predicates start with a lowercase letter, variables start with an uppercase letter, and all contain only alphanumeric characters and underscore. The TPTP language also supports interpreted symbols, which either start with a `$`, or are composed of non-alphanumeric characters. The basic logical connectives are `!`, `?`,`~`, `|`, `&`, `=>`, `<=`, `<=>`, and `<~>`, for $\forall$, $\exists$,$\neg$, $\vee$, $\wedge$, $\Rightarrow$, $\Leftarrow$, $\Leftrightarrow$, and $\oplus$ respectively. Quantified variables follow the quantifier in square brackets, with a colon to separate the quantification from the logical formula. The *source* is an optional term describing where the formula came from, e.g., an input file or an inference. The *useful_info* is an optional list of terms from user applications.

An `include` directive may include an entire file, or may specify the names of the annotated formulae that are to be included from the file. Comments in the TPTP language extend from a `%` character to the end of the line, or may be block comments within `/* ...*/` bracketing.

2.2 Higher-Order Logic

There are many quite different frameworks that fall under the general label "higher-order". The notion reaches back to Frege's original predicate calculus [11]. Inconsistencies in Frege's system, caused by the circularity of constructions such as "the set of all sets that do not contain themselves", made it clear that the expressivity of the language had to be restricted in some way. One line of development, which became the traditional route for mathematical logic, and which is not addressed further here, is the development of axiomatic first-order set theories, e.g. Zermelo-Fraenkel set theory [32].

Russell suggested using type hierarchies, and worked out ramified type theory. Church (inspired by work of Carnap) later introduced simple type theory [9], a higher-order framework built on his simply typed λ calculus, employing types to reduce expressivity and to remedy paradoxes and inconsistencies. Church's simple type theory was later extended and refined in various ways, e.g., by Martin Löf who added type universes, and by Girard and Reynolds who introduced type systems with polymorphism. This stimulated much further research in intuitionistic and constructive type theory.

Variants and extensions of Church's simple type theory have been the logic of choice for interactive proof assistants such as HOL4 [13], HOL Light [15], PVS [22], Isabelle/HOL [21], and OMEGA [25]. Church's simple type theory is also a common basis for higher-order ATP systems. The TPS system [1], which is based on a higher-order mating calculus, is a pioneering ATP system for Church's simple type theory. More recently developed higher-order ATP systems, based on extensional higher-order resolution, are LEO [5] and LEO-II [8]. Otter-λ [2] is an extension of the first-order system Otter to an untyped variant of Church's type theory. This common use of Church's simple type theory motivates using it as the starting point for THF0. For the remainder of this paper "higher-order" is therefore synonymous with Church's simple type theory [4].

A major application area of higher-order logic is hardware and software verification. Several interactive higher-order proof assistants put an emphasis on this. For example, the Isabelle/HOL system has been applied in the Verisoft project [12]. The formal proofs to be delivered in such a project require a large number of user interactions – a resource intensive task to be carried out by highly trained specialists. Often only a few of the interaction steps really require human ingenuity, and many of them could be avoided through better automation support. There are several barriers that hamper the application of automated higher-order ATP systems in this sense: (i) the available higher-order ATP systems are not yet optimized for this task; (ii) typically there are large syntax gaps between the higher-order representation languages of the proof assistants, and the input languages of the higher-order ATP systems; (iii) the results of the higher-order ATP systems have to be correctly interpreted and (iv) their proof objects might need to be translated back. The development of a commonly accepted infrastructure, and increased automation in higher-order ATP, will benefit these applications and reduce user interaction costs. Another promising application area is knowledge based reasoning. Knowledge based projects such as Cyc [19] and SUMO [20] contain a significant fraction of higher-order constructs. Reasoning in and about these knowledge sources will benefit from improved higher-order ATP systems. A strong argument for adding higher-order ATP to this application area is its demand for natural and human consumable problem and solution representations, which are harder to achieve after translating higher-order content into less expressible frameworks such as first-order logic. Further application areas of higher-order logic include computer-supported mathematics [24], and reasoning within and about multimodal logics [6].

2.3 Church's Simple Type Theory

Church's simple type theory is based on the simply typed λ calculus.

The set of simple types is freely generated from basic types ι (written `$i` in THF0) and o (`$o` in THF0), and possibly further base types using the function type constructor $\to$ (`>` in THF0).

Higher-order terms are built up from simply typed variables (X_α), simply typed constants (c_α), λ-abstraction, and application. It is assumed that sufficiently many special logical constants are available, so that all other logical connectives can be defined. For example, it is assumed that $\neg_{o\to o}$, $\vee_{o\to o\to o}$, and $\Pi_{(\alpha\to o)\to o}$ (for all simple types α) are given. The semantics of these logical symbols is fixed according to their intuitive meaning.

Well-formed (simply typed) higher-order terms are defined simultaneously for all simple types α. Variables and constants of type α are well-formed terms of type α. Given a variable X of type α and a term T of type β, the abstraction term $\lambda X.T$ is well-formed and of type $\alpha \to \beta$. Given terms S and T of types $\alpha \to \beta$ and α respectively, the application term $(S\ T)$ is well-formed and of type β.

The initial target semantics for THF0 is Henkin semantics [4,16]. However, there is no intention to fix the semantics within THF0. THF0 is designed to express problems syntactically, with the semantics being specified separately [3].

3 The THF0 Language

THF0 is a syntactically conservative extension of the untyped first-order TPTP language, adding the syntax for higher-order logic. Maintaining a consistent style between the first-order and higher-order languages facilitates easy adoption of the new language, through reuse or adaptation of existing infrastructure for processing TPTP format data, e.g., parsing tools, pretty-printing tools, system testing, and result analysis, (see Section 5). A particular feature of the TPTP language, which has been maintained in THF0, is Prolog compatibility. This allows an annotated formula to be read with a single Prolog `read/1` call, in the context of appropriate operator definitions. There are good reasons for maintaining Prolog compatibility [27].

Figure 1 presents an example problem encoded in THF0. The example is from the domain of basic set theory, stating the distributivity of union and intersection. It is a higher-order version of the first-order TPTP problem `SET171+3`, which employs first-order set theory to achieve a first-order encoding suitable for first-order theorem provers. The encoding in THF0 exploits the fact that higher-order logic provides a naturally built-in set theory, based on the idea of identifying sets with their characteristic functions. The higher-order encoding can be solved more efficiently than the first-order encoding [7,8].

The first three annotated formulae of the example are *type declarations* that declare the type signatures of $\in$, $\cap$, and $\cup$. The `type` role is new, added to the TPTP language for THF0.[2] A simple type declaration has the form *constant_symbol*: *signature*. For example, the type declaration

```
thf(const_in,type,(
    in: ( $i > ( $i > $o ) > $o ) )).
```

declares the symbol `in` (for $\in$), to be of type $\iota \to (\iota \to o) \to o$. Thus `in` expects an element of type ι and a set of type $\iota \to o$ as its arguments. The mapping arrow is right associative. Type declarations use rules starting at `<thf_typed_const>` in the BNF. Note the use of the TPTP interpreted symbols `$i` and `$o` for the standard types ι and o, which are built in to THF0. In addition to `$i` and `$o`, THF0 has the defined type `$tType` that denotes the collection of all types, and `$iType`/`$oType` as synonyms for `$i`/`$o`. Further base types can be introduced (as `<system_type>`s) on the fly. For example, the following introduces the base type u together with a corresponding constant symbol `in_u` of type $u \to (u \to o) \to o$.

```
thf(type_u,type,(
    u: $tType)).

thf(const_in_u,type,(
    in_u: ( u > ( u > $o ) > $o ) )).
```

THF0 does not support polymorphism, product types or dependent types – such language constructs are addressed in THF1.

The next three annotated formulae of the example are *axioms* that specify the meanings of $\in$, $\cap$, and $\cup$. A THF0 logical formula can use the basic TPTP connectives, λ-abstraction using the `^` quantifier followed by a list of typed λ-bound variables in square brackets, and function application using the `@` connective. Additionally, universally and existentially quantified variables must be typed. For example, the axiom

```
thf(ax_in,axiom,(
    ( in
    = ( ^ [X: $i,S: ( $i > $o )] :
          ( S @ X ) ) ) )).
```

[2] It will also be used in the forthcoming typed first-order form (TFF) TPTP language.

```
%------------------------------------------------------------------------
%----Signatures for basic set theory predicates and functions.
thf(const_in,type,(
    in: $i > ( $i > $o ) > $o  )).

thf(const_intersection,type,(
    intersection: ( $i > $o ) > ( $i > $o ) > ( $i > $o ) ) )).

thf(const_union,type,(
    union: ( $i > $o ) > ( $i > $o ) > ( $i > $o ) )).

%----Some axioms for basic set theory. These axioms define the set
%----operators as lambda-terms. The general idea is that sets are
%----represented by their characteristic functions.
thf(ax_in,axiom,(
    ( in
    = ( ^ [X: $i,S: ( $i > $o )] :
          ( S @ X ) ) ) )).

thf(ax_intersection,axiom,(
    ( intersection
    = ( ^ [S1: ( $i > $o ),S2: ( $i > $o ),U: $i] :
          ( ( in @ U @ S1 )
          & ( in @ U @ S2 ) ) ) ) )).

thf(ax_union,axiom,(
    ( union
    = ( ^ [S1: ( $i > $o ),S2: ( $i > $o ),U: $i] :
          ( ( in @ U @ S1 )
          | ( in @ U @ S2 ) ) ) ) )).

%----The distributivity of union over intersection.
thf(thm_distr,conjecture,(
    ! [A: ( $i > $o ),B: ( $i > $o ),C: ( $i > $o )] :
      ( ( union @ A @ ( intersection @ B @ C ) )
      = ( intersection @ ( union @ A @ B ) @ ( union @ A @ C ) ) ) )).
%------------------------------------------------------------------------
```

Fig. 1. Distribution of Union over Intersection, encoded in THF0

specifies the meaning of the constant $\in$ by equating it to $\lambda X_\iota.\lambda S_{\iota\to o}.(S\,X)$. Thus elementhood of an element X in a set S is reduced to applying the set S (seen as its characteristic function S) to X. Using `in`, intersection is then equated to $\lambda S^1_{\iota\to o}.\lambda S^2_{\iota\to o}.\lambda U_\iota.(\in\; U\,S^1) \wedge (\in\; U\,S^2)$, and union is equated to $\lambda S^1_{\iota\to o}.\lambda S^2_{\iota\to o}.\lambda U_\iota.(\in\; U\,S^1) \vee (\in\; U\,S^2)$. Logical formulae use rules starting at `<thf_logic_formula>` in the BNF. The explicit function application operator `@` is necessary for parsing function application expressions in Prolog. Function application is left associative.

The last annotated formula of the example is the *conjecture* to be proved.

```
thf(thm_distr,conjecture,(
    ! [A: ( $i > $o ),B: ( $i > $o ),C: ( $i > $o )] :
      ( ( union @ A @ ( intersection @ B @ C ) )
      = ( intersection @ ( union @ A @ B )
        @ ( union @ A @ C ) ) ) )).
```

This encodes $\forall A_{\iota\to o}.\forall B_{\iota\to o}.\forall C_{\iota\to o}.(A \cup (B \cap C)) = ((A \cup B) \cap (A \cup C))$. It uses rules starting at `<thf_quantified_formula>` in the BNF.

An advantage of higher-order logic over first-order logic is that the $\forall$ and $\exists$ quantifiers can be encoded using higher-order abstract syntax: Quantification is expressed using the logical constant symbols Π and Σ in connection with λ-abstraction. Higher-order systems typically make use of this feature in order to avoid introducing and supporting binders in addition to λ. THF0 provides the logical constants `!!` and `??` for Π and Σ respectively, and leaves use of the universal and existential quantifiers open to the user. Here is an encoding of the conjecture from the example, using `!!` instead of `!`.

```
thf(thm_distr,conjecture,(
    !! ( ^ [A: $i > $o,B: $i > $o,C: $i > $o] :
           ( ( union @ A @ ( intersection @ B @ C ) )
           = ( intersection @ ( union @ A @ B )
             @ ( union @ A @ C ) ) ) ) )).
```

Figure 2 presents the axioms from another example problem encoded in THF0. The example is from the domain of puzzles, and it encodes the following 'Knights and Knaves Puzzle':[3] *A very special island is inhabited only by knights and knaves. Knights always tell the truth. Knaves always lie. You meet two inhabitants: Zoey and Mel. Zoey tells you that Mel is a knave. Mel says, 'Neither Zoey nor I are knaves.' Can you determine who is a knight and who is a knave?* Puzzles of this kind have been discussed extensively in the AI literature. Here, we illustrate that an intuitive and straightforward encoding can be achieved in THF0. This encoding embeds formulas (terms of Boolean type) in terms and quantifies over variables of Boolean type; see for instance the `kk_6_2` axiom.

4 Type Checking THF0

The THF0 syntax provides a lot of flexibility. As a result, many syntactically correct expressions are meaningless because they are not well typed. For example, it does not make sense to say `& = ~` because the equated expressions have different types. It is therefore necessary to type check expressions using an inference system. Representative typing rules are given in Fig. 3 (the missing rules are obviously analogous to those provided). Here `$tType` denotes the collection of all types, and `S :: A` denotes the judgement that `S` has type `A`.

The typing rules serve only to provide an intuition. The normative definition is given by representing THF0 in the logical framework LF [14]. LF is a dependent

[3] This puzzle was auto-generated by a computer program, written by Zac Ernst.

```
%------------------------------------------------------------------
%----A very special island is inhabited only by knights and knaves.
thf(kk_6_1,axiom,(
    ! [X: $i] :
      ( ( is_a @ X @ islander )
     => ( ( is_a @ X @ knight )
        | ( is_a @ X @ knave ) ) ) )).

%----Knights always tell the truth.
thf(kk_6_2,axiom,(
    ! [X: $i] :
      ( ( is_a @ X @ knight )
     => ( ! [A: $o] :
            ( says @ X @ A )
       => A ) ) )).

%-----Knaves always lie.
thf(kk_6_3,axiom,(
    ! [X: $i] :
      ( ( is_a @ X @ knave )
     => ( ! [A: $o] : ( says @ X @ A )
       => ~ A ) ) )).

%----You meet two inhabitants: Zoey and Mel.
thf(kk_6_4,axiom,
    ( ( is_a @ zoey @ islander )
    & ( is_a @ mel @ islander ) )).

%----Zoey tells you that Mel is a knave.
thf(kk_6_5,axiom,
    ( says @ zoey @ ( is_a @ mel @ knave ) )).

%----Mel says, 'Neither Zoey nor I are knaves.'
thf(kk_6_6,axiom,
    ( says @ mel
    @ ~ ( ( is_a @ zoey @ knave )
        | ( is_a @ mel @ knave ) ) )).

%----Can you determine who is a knight and who is a knave?
thf(query,theorem,(
    ? [Y: $i,Z: $i] :
      ( ( Y = knight <~> Y = knave )
      & ( Z = knight <~> Z = knave )
      & ( is_a @ mel @ Y )
      & ( is_a @ zoey @ Z ) ) )).
%------------------------------------------------------------------
```

Fig. 2. A 'Knights and Knaves Puzzle' encoded in THF0

$$\frac{}{\texttt{\$i :: \$tType}} \qquad \frac{}{\texttt{\$o :: \$tType}} \qquad \frac{}{\Gamma \vdash \texttt{\$true :: \$o}} \qquad \frac{\texttt{thf(\_,type,c : A)}}{\Gamma \vdash \texttt{c :: A}}$$

$$\frac{\texttt{A :: \$tType} \qquad \texttt{B :: \$tType}}{\texttt{A > B :: \$tType}} \qquad \frac{\texttt{X :: A} \text{ in } \Gamma}{\Gamma \vdash \texttt{X :: A}}$$

$$\frac{\Gamma, \texttt{X :: A} \vdash \texttt{S :: B}}{\Gamma \vdash \texttt{\^{} [X : A] : S :: A > B}} \qquad \frac{\Gamma \vdash \texttt{F :: A > B} \qquad \Gamma \vdash \texttt{S :: A}}{\Gamma \vdash \texttt{F @ S :: B}}$$

$$\frac{\Gamma \vdash \texttt{F :: A > \$o}}{\Gamma \vdash \texttt{!! F :: \$o}} \qquad \frac{\Gamma, \texttt{X :: A} \vdash \texttt{F :: \$o}}{\Gamma \vdash \texttt{! [X : A]: F :: \$o}} \qquad \frac{\Gamma \vdash \texttt{F :: \$o}}{\Gamma \vdash \texttt{\~{} F :: \$o}}$$

$$\frac{\Gamma \vdash \texttt{F :: \$o} \qquad \Gamma \vdash \texttt{G :: \$o}}{\Gamma \vdash \texttt{F \& G :: \$o}} \qquad \frac{\Gamma \vdash \texttt{S :: A} \qquad \Gamma \vdash \texttt{S' :: A}}{\Gamma \vdash \texttt{S = S' :: \$o}}$$

Fig. 3. Typing rules of THF0

type theory related to Martin-Löf type theory [18], particularly suited to logic representations. In particular, the maintenance of the context, substitution, and α-conversion are handled by LF and do not have to be specified separately. A THF0 expression is well typed if its translation to LF is, and THF0 expressions can be type checked using existing tools such as Twelf [23]. In the following, Twelf syntax is used to describe the representation.

To represent THF0, a base signature Σ_0 is defined, as given in Figure 4. It is similar to the Harper et al. encoding of higher-order logic [14]. Every list L of TPTP formulas can be translated to a list of declarations Σ_L extending Σ_0. A list L is well-formed iff Σ_0, Σ_L is a well-formed Twelf signature. The signature of the translation is given in Figure 5.

In Figure 4, `$tType : type` means that the name `$tType` is introduced as an LF type. Its LF terms are the translations of THF0 types. The declaration `$tm : $tType -> type` introduces a symbol `$tm`, which has no analog in THF0: for every THF0 type `A` it declares an LF type `$tm A`, which holds the terms of type `A`. `$i` and `$o` are declared as base types, and `>` is the function type constructor. Here `->` is the LF symbol for function spaces, i.e., the declaration of `>` means that it takes two types as arguments and returns a type. Thus, on the ASCII level, the translation from THF0 types to LF is the identity. The symbols `^` through `??` declare term and formula constructors with their types. The binary connectives other than `&` have been omitted - they are declared just like `&`. In Twelf, equality and inequality are actually ternary symbols, because they take the type `A` of the equated terms as an additional argument. By using implicit arguments, Twelf is able to reconstruct `A` from the context, so that equality behaves essentially like a binary symbol. Except for equality, the same ASCII notation can be used in Twelf as in THF0 so that on the ASCII level most cases of the translation are trivial. The translation of the binders `^`, `!`, and `?` is interesting: they are all expressed by the λ binder of Twelf. For example, if F is translated to F', then $\forall x_{\$i}.F$ is translated to $!(\lambda x_{\$\,tm\ \$i}.F')$. Since the Twelf ASCII syntax for $\lambda x_{\$\,tm\ \$i}.F'$ is simply `[x:$tm $i] F'`, on the ASCII level the translation of binders consists of simply inserting `$tm` and dropping a colon. For

```
$tType  : type.
$tm     : $tType -> type.
$i      : $tType.
$o      : $tType.
>       : $tType -> $tType -> $tType.

^       : ($tm A -> $tm B) -> $tm (A > B).
@       : $tm(A > B) -> $tm A -> $tm B.
$true   : $tm $o.
$false  : $tm $o.
~       : $tm $o -> $tm $o.
&       : $tm $o -> $tm $o -> $tm $o.
==      : $tm A -> $tm A -> $tm $o.
!=      : $tm A -> $tm A -> $tm $o.

!       : ($tm A -> $tm $o) -> $tm $o.
?       : ($tm A -> $tm $o) -> $tm $o.
!!      : ($tm(A > $o)) -> $tm $o.
??      : ($tm(A > $o)) -> $tm $o.

$istrue : $tm $o -> type.
```

Fig. 4. The base signature for Twelf

THF0	LF
types	terms of type `$tType`
terms of type `A`	terms of type `$tm A`
formulas	terms of type `$tm $o`

Fig. 5. Correspondences between THF0 and the LF representation

all binders, the type `A` of the bound variable is an implicit argument and thus reconstructed by Twelf. Of course, most of the term constructors can be defined in terms of only a few primitive ones. In particular, the Twelf symbol `!` can be defined as the composition of `!!` and `^`, and similarly `?`. However, since they are irrelevant for type checking, definitions are not used here.

Figure 6 shows the translation of the formulae from Figure 1. A THF0 type declaration for a constant `c` with type `A` is translated to the Twelf declaration `c : $tm A`. An axiom or conjecture `F` is translated to a Twelf declaration for the type `$istrue F'`, where `F'` is the translation of `F`. In the latter case the Twelf name of the declaration is irrelevant for type-checking. By using the role, i.e., axiom or conjecture, as the Twelf name, the role of the original TPTP formula can be recovered from its Twelf translation.

Via the Curry-Howard correspondence [10,17], the representation of THF0 in Twelf can be extended to represent proofs – it is necessary only to add declarations for the proof rules to Σ_0, i.e., β-η-conversion, and the rules for the connectives and quantifiers. If future versions of THF0 provide a standard format for explicit proof terms, Twelf can serve as a neutral trusted proof checker.

```
in           : $tm($i > ($i > $o) > $o).
intersection : $tm(($i > $o) > ($i > $o) > ($i > $o)).
union        : $tm(($i > $o) > ($i > $o) > ($i > $o)).
axiom        : $istrue in == ^[X : $tm $i] ^[S : $tm($i > $o)](S @ X).
axiom        : $istrue intersection ==
                    ^[S1: $tm($i > o)] ^[S2: $tm($i > $o)] ^[U: $tm $i]
                     ((in @ U @ S1) & (in @ U @ S2)).
axiom        : $istrue union ==
                    ^[S1: $tm($i > $o)] ^[S2: $tm($i > $o)] ^[U: $tm $i]
                     (in @ U @ S1) | (in @ U @ S2).
conjecture   : $istrue
              !![A: $tm($i > $o)] !![B: $tm($i > $o)] !![C: $tm($i > $o)]
                 ((union @ A @ (intersection @ B @ C))
                 == (intersection @ (union @ A @ B) @ (union @ A @ C))).
```

Fig. 6. Twelf translation of Figure 1

5 TPTP Resources

The first-order TPTP provides a range of resources to support use of the problem library and related infrastructure. Many of these resources are immediately applicable to the higher-order setting, while some require changes to reflect the new features in the THF0 language.

The main resource for users is the TPTP problem library itself. At the time of writing around 100 THF problems have been collected, and are being prepared for addition to a THF extension of the library. THF file names have an `@` separator between the abstract problem name and the version number (corresponding to the `-` in CNF problem names and the `+` in FOF problem names), e.g., the example problem in Figure 1 will be put in `SET171@4.p`. The existing header fields of TPTP problems have been slightly extended to deal with higher-order features. First, the `Status` field, which records the semantic status of the problem in terms of the SZS ontology, provide status values for each semantics of interest [3,4]. Second, the `Syntax` field, which records statistics of syntactic characteristics of the problem, has been extended to include counts of the new connectives that are part of THF0.

Tools that support the TPTP include the tptp2X and tptp4X utilities, which read, analyse, transform, and output TPTP problems. tptp2X is written in Prolog, and it has been extended to read, analyse, and output problems written in THF0. The translation to Twelf syntax has been implemented as a tptp2X format module, producing files that can be type-checked directly. The Twelf translation has been used to type-check (and in some cases correct) the THF problems collected thus far. The extended tptp2X is part of TPTP v3.4.0. tptp4X is based on the `JJParser` library of code written in C. This will be extended to cope with higher-order formulae.

The flipside of the TPTP problem library is the TSTP solution library. Once the higher-order part of the TPTP problem library is in place it is planned to extend the TSTP to include results from higher-order ATP systems on the THF problems. The harnesses used for building the first-order TSTP will be used as-is for the higher-order extension.

A first competition "happening" for higher-order ATP systems that can read THF0 will be held at IJCAR 2008. This event will be similar to the CASC competition for first-order ATP systems, but with a less formal evaluation phase. It will exploit and test the THF0 language.

6 Conclusion

This paper has described, with examples, the core of the TPTP language for higher-order logic (Church's simple type theory) – THF0. The TPTP infrastructure is being extended to support problems, solutions, tools, and evaluation of ATP systems using the THF0 language. Development and adoption of the THF0 language and associated TPTP infrastructure will support research and development in automated higher-order reasoning, providing leverage for progress leading to effective and successful application. To date only the LEO II system [8] is known to use the THF0 language. It is hoped that more high-order reasoning systems and tools will adopt the THF language, making easy and direct communication between the systems and tools possible.

Acknowledgements. Chad Brown contributed to the design of the THF language. Allen Van Gelder contributed to the development of the THF syntax and BNF. Frank Theiss and Arnaud Fietzke implemented the LEO-II parser for THF0 language and provided useful feedback.

References

1. Andrews, P.B., Bishop, M., Issar, S., Nesmith, D., Pfenning, F., Xi, H.: TPS: A Theorem-Proving System for Classical Type Theory. Journal of Automated Reasoning 16(3), 321–353 (1996)
2. Beeson, M.: Otter-lambda, a Theorem-prover with Untyped Lambda-unification. In: Sutcliffe, G., Schulz, S., Tammet, T. (eds.) Proceedings of the Workshop on Empirically Successful First Order Reasoning (2004)
3. Benzmüller, C., Brown, C.: A Structured Set of Higher-Order Problems. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 66–81. Springer, Heidelberg (2005)
4. Benzmüller, C., Brown, C., Kohlhase, M.: Higher-order Semantics and Extensionality. Journal of Symbolic Logic 69(4), 1027–1088 (2004)
5. Benzmüller, C., Kohlhase, M.: LEO - A Higher-Order Theorem Prover. In: Kirchner, C., Kirchner, H. (eds.) CADE 1998. LNCS (LNAI), vol. 1421, pp. 139–143. Springer, Heidelberg (1998)
6. Benzmüller, C., Paulson, L.: Exploring Properties of Normal Multimodal Logics in Simple Type Theory with LEO-II. In: Benzmüller, C., Brown, C., Siekmann, J., Statman, R. (eds.) Festschrift in Honour of Peter B. Andrews on his 70th Birthday. IfCoLog (to appear 2007)
7. Benzmüller, C., Sorge, V., Jamnik, M., Kerber, M.: Combined Reasoning by Automated Cooperation. Journal of Applied Logic (in print) (2008)

8. Benzmüller, C., Theiss, F., Paulson, L., Fietzke, A.: LEO-II - A Cooperative Automatic Theorem Prover for Higher-Order Logic. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) Proceedings of the 4th International Joint Conference on Automated Reasoning (IJCAR 2008). LNCS (LNAI), vol. 5195. Springer, Heidelberg (2008)
9. Church, A.: A Formulation of the Simple Theory of Types. Journal of Symbolic Logic 5, 56–68 (1940)
10. Curry, H.B., Feys, R.: Combinatory Logic I. North Holland, Amsterdam (1958)
11. Frege, F.: Grundgesetze der Arithmetik. Jena (1893) (1903)
12. Godefroid, P.: Software Model Checking: the VeriSoft Approach. Technical Report Technical Memorandum ITD-03-44189G, Bell Labs, Lisle, USA (2003)
13. Gordon, M., Melham, T.: Introduction to HOL, a Theorem Proving Environment for Higher Order Logic. Cambridge University Press, Cambridge (1993)
14. Harper, R., Honsell, F., Plotkin, G.: A Framework for Defining Logics. Journal of the ACM 40(1), 143–184 (1993)
15. Harrison, J.: HOL Light: A Tutorial Introduction. In: Srivas, M., Camilleri, A. (eds.) FMCAD 1996. LNCS, vol. 1166, pp. 265–269. Springer, Heidelberg (1996)
16. Henkin, L.: Completeness in the Theory of Types. Journal of Symbolic Logic 15, 81–91 (1950)
17. Howard, W.: The Formulas-as-types Notion of Construction. In: Seldin, J., Hindley, J. (eds.) H B Curry, Essays on Combinatory Logic, Lambda Calculus and Formalism, pp. 479–490. Academic Press, London (1980)
18. Martin-Löf, P.: An Intuitionistic Theory of Types. In: Sambin, G., Smith, J. (eds.) Twenty-Five Years of Constructive Type Theory, pp. 127–172. Oxford University Press, Oxford (1973)
19. Matuszek, C., Cabral, J., Witbrock, M., DeOliveira, J.: An Introduction to the Syntax and Content of Cyc. In: Baral, C. (ed.) Proceedings of the 2006 AAAI Spring Symposium on Formalizing and Compiling Background Knowledge and Its Applications to Knowledge Representation and Question Answering, pp. 44–49 (2006)
20. Niles, I., Pease, A.: Towards A Standard Upper Ontology. In: Welty, C., Smith, B. (eds.) Proceedings of the 2nd International Conference on Formal Ontology in Information Systems, pp. 2–9 (2001)
21. Nipkow, T., Paulson, L.C., Wenzel, M.T.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
22. Owre, S., Rajan, S., Rushby, J.M., Shankar, N., Srivas, M.: PVS: Combining Specification, Proof Checking, and Model Checking. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 411–414. Springer, Heidelberg (1996)
23. Pfenning, F., Schürmann, C.: System Description: Twelf - A Meta-Logical Framework for Deductive Systems. In: Ganzinger, H. (ed.) CADE 1999. LNCS (LNAI), vol. 1632, pp. 202–206. Springer, Heidelberg (1999)
24. Rudnicki, P.: An Overview of the Mizar Project. In: Proceedings of the 1992 Workshop on Types for Proofs and Programs, pp. 311–332 (1992)
25. Siekmann, J., Benzmüller, C., Autexier, S.: Computer supported mathematics with omega. Journal of Applied Logic 4(4), 533–559 (2006)
26. Sutcliffe, G.: TPTP, TSTP, CASC, etc. In: Diekert, V., Volkov, M.V., Voronkov, A. (eds.) CSR 2007. LNCS, vol. 4649, pp. 7–23. Springer, Heidelberg (2007)
27. Sutcliffe, G., Schulz, S., Claessen, K., Gelder, A.V.: Using the TPTP Language for Writing Derivations and Finite Interpretations. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 67–81. Springer, Heidelberg (2006)

28. Sutcliffe, G., Suttner, C.: The State of CASC. AI Communications 19(1), 35–48 (2006)
29. Sutcliffe, G., Suttner, C.B.: The TPTP Problem Library: CNF Release v1.2.1. Journal of Automated Reasoning 21(2), 177–203 (1998)
30. Sutcliffe, G., Zimmer, J., Schulz, S.: TSTP Data-Exchange Formats for Automated Theorem Proving Tools. In: Zhang, W., Sorge, V. (eds.) Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems. Frontiers in Artificial Intelligence and Applications, vol. 112, pp. 201–215. IOS Press, Amsterdam (2004)
31. Gelder, A.V., Sutcliffe, G.: Extending the TPTP Language to Higher-Order Logic with Automated Parser Generation. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 156–161. Springer, Heidelberg (2006)
32. Zermelo, E.: Über Grenzzahlen und Mengenbereiche. Fundamenta Mathematicae 16, 29–47 (1930)

A BNF for THF0

The BNF uses a modified BNF meta-language that separates syntactic, semantic, lexical, and character-macro rules [27]. Syntactic rules use the standard `::=` separator, semantic constraints on the syntactic rules use a `:==` separator, rules that produce tokens from the lexical level use a `::-` separator, and the bottom level character-macros are defined by regular expressions in rules using a `:::` separator. The BNF is easy to translate into parser-generator (`lex/yacc`, `antlr`, etc.) input [31].

The syntactic and semantic grammar rules for THF0 are presented here. The rules defining tokens and character macros are available from the TPTP web site, www.tptp.org.

```
%------------------------------------------------------------------------------
%----Files. Empty file is OK.
<TPTP_file>          ::= <TPTP_input>*
<TPTP_input>         ::= <annotated_formula> | <include>

%----Formula records
<annotated_formula>  ::= <thf_annotated>
<thf_annotated>      ::= thf(<name>,<formula_role>,<thf_formula><annotations>).
<annotations>        ::= <null> | ,<source><optional_info>
%----In derivations the annotated formulae names must be unique, so that
%----parent references (see <inference_record>) are unambiguous.

%----Types for problems.
<formula_role>       ::= <lower_word>
<formula_role>       :== axiom | hypothesis | definition | assumption |
                         lemma | theorem | conjecture | negated_conjecture |
                         plain | fi_domain | fi_functors | fi_predicates |
                         type | unknown
%------------------------------------------------------------------------------
%----THF0 formulae. All formulae must be closed.
<thf_formula>        ::= <thf_logic_formula> | <thf_typed_const>
<thf_logic_formula>  ::= <thf_binary_formula> | <thf_unitary_formula>
<thf_binary_formula> ::= <thf_pair_binary> | <thf_tuple_binary>
%----Only some binary connectives can be written without ()s.
%----There's no precedence among binary connectives
<thf_pair_binary>    ::= <thf_unitary_formula> <thf_pair_connective>
                         <thf_unitary_formula>
%----Associative connectives & and | are in <assoc_formula>.
```

```
<thf_tuple_binary>   ::= <thf_or_formula> | <thf_and_formula> |
                         <thf_apply_formula>
<thf_or_formula>     ::= <thf_unitary_formula> <vline> <thf_unitary_formula> |
                         <thf_or_formula> <vline> <thf_unitary_formula>
<thf_and_formula>    ::= <thf_unitary_formula> & <thf_unitary_formula> |
                         <thf_and_formula> & <thf_unitary_formula>
<thf_apply_formula>  ::= <thf_unitary_formula> @ <thf_unitary_formula> |
                         <thf_apply_formula> @ <thf_unitary_formula>
%----<thf_unitary_formula> are in ()s or do not have a <binary_connective>
%----at the top level. Essentially, a <thf_unitary_formula> is any lambda
%----expression that "has enough parentheses" to be used inside a larger
%----lambda expression. However, lambda notation might not be used.
<thf_unitary_formula> ::= <thf_quantified_formula> | <thf_abstraction> |
                         <thf_unary_formula> | <thf_atom> |
                         (<thf_logic_formula>)
<thf_quantified_formula> ::= <thf_quantified_var> | <thf_quantified_novar>
<thf_quantified_var> ::= <quantifier> [<thf_variable_list>] :
                         <thf_unitary_formula>
<thf_quantified_novar>   ::= <thf_quantifier> (<thf_unitary_formula>)
%----@ (denoting apply) is left-associative and lambda is right-associative.
<thf_abstraction>    ::= <thf_lambda> [<thf_variable_list>] :
                         <thf_unitary_formula>
<thf_variable_list>  ::= <thf_variable> | <thf_variable>,<thf_variable_list>
<thf_variable>       ::= <variable> | <thf_typed_variable>
<thf_typed_variable> ::= <variable> : <thf_top_level_type>
%----Unary connectives bind more tightly than binary. The negated formula
%----must be ()ed because a ~ is also a term.
<thf_unary_formula>  ::= <thf_unary_connective> (<thf_logic_formula>)

%----An <thf_typed_const> is a global assertion that the atom is in this type.
<thf_typed_const>    ::= <constant> : <thf_top_level_type> | (<thf_typed_const>)

%----THF atoms
<thf_atom>           ::= <constant> | <defined_constant> | <system_constant> |
                         <variable> | <thf_conn_term>
%----<defined_constant> is really a <defined_prop>, but that's the syntax rule
%----used. <thf_atom> can also be <defined_type>, but they are syntactically
%----captured by <defined_constant>. Ditto for <system_*>.

%----<thf_top_level_type> appears after ":", where a type is being specified
%----for a term or variable.
<thf_unitary_type>   ::= <constant> | <variable> | <defined_type> |
                         <system_type> | (<thf_binary_type>)
<thf_top_level_type> ::= <constant> | <variable> | <defined_type> |
                         <system_type> | <thf_binary_type>
<thf_binary_type>    ::= <thf_mapping_type> | (<thf_binary_type>)
<thf_mapping_type>   ::= <thf_unitary_type> <arrow> <thf_unitary_type> |
                         <thf_unitary_type> <arrow> <thf_mapping_type>
%------------------------------------------------------------------------------
%----Special higher order terms
<thf_conn_term>      ::= <thf_quantifier> | <thf_pair_connective> |
                         <assoc_connective> | <thf_unary_connective>

%----Connectives - THF
<thf_lambda>         ::= ^
<thf_quantifier>     ::= !! | ??
<thf_pair_connective> ::= <defined_infix_pred> | <binary_connective>
<thf_unary_connective> ::= <unary_connective>
%----Connectives - FOF
<quantifier>         ::= ! | ?
<binary_connective>  ::= <=> | => | <= | <~> | ~<vline> | ~&
<assoc_connective>   ::= <vline> | &
<unary_connective>   ::= ~

%----Types for THF and TFF
<defined_type>       ::= <atomic_defined_word>
<defined_type>       :== $oType | $o | $iType | $i | $tType
%----$oType/$o is the Boolean type, i.e., the type of $true and $false.
```

```
%----$iType/$i is type of individuals. $tType is the type of all types.
<system_type>        ::= <atomic_system_word>

%----First order atoms
<defined_prop>       :== <atomic_defined_word>
<defined_prop>       :== $true | $false
<defined_infix_pred> ::= = | !=

%----First order terms
<constant>           ::= <functor>
<functor>            ::= <atomic_word>
<defined_constant>   ::= <atomic_defined_word>
<system_constant>    ::= <atomic_system_word>
<variable>           ::= <upper_word>
%------------------------------------------------------------------------------
%----General purpose
<name>               ::= <atomic_word> | <unsigned_integer>
<atomic_word>        ::= <lower_word> | <single_quoted>
<atomic_defined_word> ::= <dollar_word>
<atomic_system_word> ::= <dollar_dollar_word>
<null>               ::=
%------------------------------------------------------------------------------
```

Focusing in Linear Meta-logic

Vivek Nigam and Dale Miller

INRIA & LIX/École Polytechnique, Palaiseau, France
nigam@lix.polytechnique.fr, dale.miller@inria.fr

Abstract. It is well known how to use an intuitionistic meta-logic to specify natural deduction systems. It is also possible to use linear logic as a meta-logic for the specification of a variety of sequent calculus proof systems. Here, we show that if we adopt different *focusing* annotations for such linear logic specifications, a range of other proof systems can also be specified. In particular, we show that natural deduction (normal and non-normal), sequent proofs (with and without cut), tableaux, and proof systems using general elimination and general introduction rules can all be derived from essentially the same linear logic specification by altering focusing annotations. By using elementary linear logic equivalences and the completeness of focused proofs, we are able to derive new and modular proofs of the soundness and completeness of these various proofs systems for intuitionistic and classical logics.

1 Introduction

Logics and type systems have been exploited in recent years as frameworks for the specification of deduction in a number of logics. The most common such *meta-logics* and *logical frameworks* have been based on intuitionistic logic (see, for example, [FM88]) or dependent types (see [HHP93, Pfe89]). Such intuitionistic logics can be used to directly encode natural deduction style proof systems.

In a series of papers [Mil96, Pim01, MP02, MP04, PM05], Miller & Pimentel used classical linear logic as a meta-logic to specify and reason about a variety of sequent calculus proof systems. Since the encodings of such logical systems are natural and direct, the meta-theory of linear logic can be used to draw conclusions about the object-level proof systems. More specifically, in [MP02], a decision procedure was presented for determining if one encoded proof system is derivable from another. In the same paper, necessary conditions were presented (together with a decision procedure) for assuring that an encoded proof system satisfies cut-elimination. This last result used linear logic's dualities to formalize the fact that if the left and right introduction rules are suitable duals of each other then non-atomic cuts can be eliminated.

In this paper, we again use linear logic as a meta-logic but make critical use of the completeness of *focused proofs* for linear logic. Roughly speaking, focused proofs in linear logic divide sequent calculus proofs into two different phases: the *negative* phase involves rules that are invertible while the *positive* phase involves the focused non-invertible rules. In linear logic, it is clear to which phase each linear logic connective appears but it is completely arbitrary how atomic formulas can be assigned to these different phases. For example, all atomic formulas can be assigned a *negative polarity* or a *positive polarity* or, in fact, any mixture of these. The completeness of focused proofs then states that if a formula B is provable in linear logic and we fix on any polarity assignment to atomic formulas, then B will have a focused proof. (Soundness also holds.) Thus, while polarity assignment does not affect provability, it can result in strikingly different proofs. The earlier works of Miller & Pimentel assumed that all atoms were given negative polarity: this resulted in an encoding of object-level sequent calculus. As we shall show here, if we vary that polarity assignment, we can get other object-level proof systems represented. Thus, while provability is not affected, different, meta-level, focused proofs are built and these encode different object-level proof systems.

Our main contribution in this paper is illustrating how a range of proof systems can be seen as different focusing disciplines on the same or (meta-logically) equivalent sets of linear logic specifications. Soundness and relative completeness are generally trivial consequences of linear logic identities. In particular, we present examples based on sequent calculus and natural deduction [Gen69], Generalized Elimination Rules [vP01], Free Deduction [Par92], the tableaux system KE [DM94], and Smullyan's Analytic Cut [Smu68]. The adequacy of a given specification of inference rules requires first assigning polarity to meta-level atoms using in the specification: then adequacy is generally an immediate consequence of the focusing theorem of linear logic.

Finally, we attempt to point out how deep the equivalence of encoded proof systems goes by describing three levels of encoding adequacy: *relative completeness* where the provable set of formulas is the same, *full completeness of proofs* where the completed proofs are in one-to-one correspondence, and *full completeness of derivations* where (open) derivations (such as inference rules themselves) are also in one-to-one correspondence.

2 Preliminaries

2.1 Linear Logic

We shall assume that the reader is familiar with linear logic. We review a few basic points here. *Literals* are either atomic formulas or their negations. We write $\neg F$ to denote the *negation normal form* of the formula F: that is, formulas computed by using de Morgan dualities and where negation has only atomic scope. The connectives $\otimes$ and $\parr$ and their units 1 and $\bot$ are *multiplicative*; the connectives $\oplus$ and $\&$ and their units 0 and $\top$ are *additive* connectives; $\forall$ and $\exists$ are (first-order) quantifiers; and $!$ and $?$ are the exponentials.

In general, we shall present *theories* in the linear meta-logic as appearing on the right-hand side of sequents. Thus, if $\mathcal{X}$ is a set of closed formulas then we say that the formula B is derived using theory $\mathcal{X}$ if $\vdash B, \mathcal{X}$ is provable in linear logic. We shall also write $B \equiv C$ to denote the formula $(\neg B \parr C) \& (\neg C \parr B)$.

$(\Rightarrow_L)$	$\lfloor A \Rightarrow B \rfloor^\perp \otimes (\lceil A \rceil \otimes \lfloor B \rfloor)$	$(\Rightarrow_R)$	$\lceil A \Rightarrow B \rceil^\perp \otimes (\lfloor A \rfloor \parr \lceil B \rceil)$
$(\wedge_L)$	$\lfloor A \wedge B \rfloor^\perp \otimes (\lfloor A \rfloor \oplus \lfloor B \rfloor)$	$(\wedge_R)$	$\lceil A \wedge B \rceil^\perp \otimes (\lceil A \rceil \mathbin{\&} \lceil B \rceil)$
$(\vee_L)$	$\lfloor A \vee B \rfloor^\perp \otimes (\lfloor A \rfloor \mathbin{\&} \lfloor B \rfloor)$	$(\vee_R)$	$\lceil A \vee B \rceil^\perp \otimes (\lceil A \rceil \oplus \lceil B \rceil)$
$(\forall_L)$	$\lfloor \forall B \rfloor^\perp \otimes \lfloor Bx \rfloor$	$(\forall_R)$	$\lceil \forall B \rceil^\perp \otimes \forall x \lceil Bx \rceil$
$(\exists_L)$	$\lfloor \exists B \rfloor^\perp \otimes \forall x \lfloor Bx \rfloor$	$(\exists_R)$	$\lceil \exists B \rceil^\perp \otimes \lceil Bx \rceil$
$(\perp_L)$	$\lfloor \perp \rfloor^\perp$	(t_R)	$\lceil t \rceil^\perp \otimes \top$

Fig. 1. The theory $\mathcal{L}$ used to encode various proof systems for minimal, intuitionistic, and classical logics

(Id_1)	$\lfloor B \rfloor^\perp \otimes \lceil B \rceil^\perp$	(Id_2)	$\lfloor B \rfloor \otimes \lceil B \rceil$	(W_R)	$\lceil C \rceil^\perp \otimes \perp$
(Str_L)	$\lfloor B \rfloor^\perp \otimes ?\lfloor B \rfloor$	(Str_R)	$\lceil B \rceil^\perp \otimes ?\lceil B \rceil$		

Fig. 2. Specification of the identity rules (cut and initial) and of the structural rules (weakening and contraction)

2.2 Encoding Object-Logic Formulas, Sequents, and Inference Rules

We use linear logic as a meta-logic to encode object logics, in a similar fashion as done in [Mil96, Pim01]. We shall assume that our meta-logic is a multi-sorted version of linear logic: in particular, the type o denotes meta-level formulas, the type *form* denotes object-level formulas, and the type i denotes object-level terms. Object-level formulas are encoded in the usual way: in particular, the object-level quantifiers $\forall$, $\exists$ are given the type $(i \rightarrow form) \rightarrow form$ and the expressions $\forall(\lambda x.B)$ and $\exists(\lambda x.B)$ are written, respectively, as $\forall x.B$ and $\exists x.B$. To deal with quantified object-level formulas, our meta-logic will quantify over variables of types $i \rightarrow \cdots \rightarrow i \rightarrow form$ (for 0 or more occurrences of i).

Encoding object-level sequents as meta-logic sequents is done by introducing two meta-level predicates of type $form \rightarrow o$, written as $\lfloor \cdot \rfloor$ and $\lceil \cdot \rceil$, and then writing the two-sided, object-level sequent $B_1, \ldots, B_n \vdash C_1, \ldots, C_m$ as the one-sided, meta-level sequent $\vdash \lfloor B_1 \rfloor, \ldots, \lfloor B_n \rfloor, \lceil C_1 \rceil, \ldots, \lceil C_m \rceil$. Thus formulas on the left of the object-level sequent are marked using $\lfloor \cdot \rfloor$ and formulas on the right of the object-level sequent are marked using $\lceil \cdot \rceil$. We shall assume that object-level sequents are pairs of either sets or multisets and that meta-level sequents are multisets of formulas. For convenience, if Γ is a (multi)set of formulas, $\lfloor \Gamma \rfloor$ (resp. $\lceil \Gamma \rceil$) denotes the multiset of atoms $\{\lfloor F \rfloor \mid F \in \Gamma\}$ (resp. $\{\lceil F \rceil \mid F \in \Gamma\}$).

Inference rules generally attribute to a logical connective two dual "senses": in sequent calculus, these correspond to the left-introduction and right-introduction rules while in natural deduction, these correspond to the introduction and elimination rules. Consider the linear logic formulas in Figure 1. When we display formulas in this manner, we intend that the named formula is actually the result of applying ? to existential closure of the formula. Thus, the formula named $\wedge_L$ is actually $?\exists A \exists B[\lfloor A \wedge B \rfloor^\perp \otimes (\lfloor A \rfloor \oplus \lfloor B \rfloor)]$. The formulas in Figure 1 help to provide the meaning of connectives in a rather abstract and succinct fashion. For example, the conjunction connective appears in two formulas: once in the scope of $\lfloor \cdot \rfloor$ and once in the scope of $\lceil \cdot \rceil$. Notice that there is no explicit reference to side formulas or any side conditions for any of these rules. We shall provide a much more in-depth analysis of the formulas in Figure 1 in the following sections.

The formulas in Figure 2 play a central role in this paper. The Id_1 and Id_2 formulas can prove the duality of the $\lfloor\cdot\rfloor$ and $\lceil\cdot\rceil$ predicates: in particular, one can prove in linear logic that

$$\vdash \forall B(\lceil B\rceil \equiv \lfloor B\rfloor^{\perp}) \mathbin{\&} \forall B(\lfloor B\rfloor \equiv \lceil B\rceil^{\perp}), Id_1, Id_2$$

Similarly, the formulas Str_L and Str_R allow us to prove the equivalences $\lfloor B\rfloor \equiv ?\lfloor B\rfloor$ and $\lceil B\rceil \equiv ?\lceil B\rceil$. The last two equivalences allows the weakening and contraction of formulas at both the meta-level and object-level. For instance, in the encoding of minimal logics, where structural rules are only allowed in the left-hand-side, one should include only the Str_L formula; while in the encoding of classical logics, where structural rules are allowed in both sides of a sequent, one should include both Str_L and Str_R formulas. Moreover, since the presence of these two formulas allows contracting and weakening of $\lfloor\cdot\rfloor$ and $\lceil\cdot\rceil$ atoms, one can show that the specification $\mathcal{L}\cup\{Str_L, Str_R\}$ is equivalent to the specification obtained from it but where the "additive rules" $\wedge_L, \wedge_R, \vee_L, \vee_R$ are replaced by the existential closure of their multiplicative versions, namely

$$\begin{array}{ll}\lceil A\wedge B\rceil^{\perp}\otimes(\lceil A\rceil\otimes\lceil B\rceil) & \lfloor A\wedge B\rfloor^{\perp}\otimes(\lfloor A\rfloor \parr \lfloor B\rfloor)\\ \lfloor A\vee B\rfloor^{\perp}\otimes(\lceil A\rceil\otimes\lceil B\rceil) & \lceil A\vee B\rceil^{\perp}\otimes(\lceil A\rceil \parr \lfloor B\rfloor).\end{array}$$

The formula W_R encodes the weakening right rule and is used to encode intuitionistic logics, where weakening, but not contraction, is allowed on formulas on the right-hand-side of a sequent.

2.3 Adequacy Levels for Encodings

When comparing deductive systems, one can easily identify several "levels of adequacy". For example, Girard in [Gir06, Chapter 7] proposes three levels of adequacy based on semantical notions: the level of *truth*, the level of *functions*, and the level of *actions*. Here, we also identify three levels of adequacy but from a proof-theoretical point-of-view. The weakest level of adequacy is *relative completeness* which considers only *provability*: a formula has a proof in one system if it has a proof in another system. A stronger level of adequacy is of *full completeness of proofs*: the proofs of a given formula are in one-to-one correspondence with proofs in another system. If one uses the term "derivation" for possibly incomplete proofs (proofs that may have open premises), we can consider a even stronger level of adequacy. We use the term *full completeness of derivations*, if the derivations (such as inference rules themselves) in one system are in one-to-one correspondence with those in another system. When we state equivalences between proof systems (usually between object-level proof systems and their meta-level encoding), we will often comment on which level the theorem should be placed.

2.4 A Focusing Proof System for Linear Logic

In [And92], Andreoli proved the completeness of the focused proof system for linear logic given in Figure 3. Focusing proof systems involve applying inference rules in alternating *polarities* or phases. In particular, formulas are *negative* if their top-level connective is either $\parr, \bot, \&, \top, ?,$ or $\forall$; formulas are *positive* if their top-level connective is $\oplus, 0, \otimes, 1, !,$ or $\exists$. This polarity assignment is rather natural in the sense that all right introduction rules for negative formulas are invertible while such introduction rules for positive formulas are not necessarily invertible. The only formulas that are not given polarities by the above assignment are the literals. In the negative phase, represented by the judgment, $\vdash \Theta : \Gamma \Uparrow L$, rules are applied only to negative formulas appearing in L, while positive formulas are moved to one of the multisets, Θ or Γ, to the left of the $\Uparrow$, by using the $[R \Uparrow]$ or $[?]$ rules. When L is empty the positive phase begins by using one of the decide rules, $[D_1]$ or $[D_2]$, and selecting one formula to focus on, represented by the judgment $\vdash \Theta : \Gamma \Downarrow F$. Rules are then applied hereditarily to subformulas of F, but if a negative subformula is encountered, focus is lost by using the reaction rule $[R \Downarrow]$ and another negative phase begins. Andreoli's completeness theorem can be interpreted as follows: If F is a provable linear logic formula, then for *any assignment* of polarities to the atomic formulas of linear logic, the sequent $\vdash \cdot : \cdot \Uparrow F$ is provable.

We point out two important aspects of this completeness theorem. First, the focus proof system only works on "annotated formulas" and not regular formulas. Here, the annotation is a mapping of atoms to polarities. (In intuitionistic and classical logics, one may also need to annotate conjunctions and disjunctions [LM07].) Notice that the rules $[I_1]$ and $[I_2]$ explicitly refer to the polarity assigned to literals. Second, an annotation does not affect *provability* but it may affect greatly the structure of (focused) proofs that are possible. In papers such as [LM07, MN07], differences in annotations allowed one to build only top-down (goal-directed) proofs or only bottom-up (program-directed) proofs or combinations of both. In this paper, we shall illustrate how it is possible to use different polarity assignments (in the linear meta-logic) to derive different proof systems (of an object-logic). In particular, sequent calculus and natural deduction can be seen as *two different* annotations of the *same* linear logic specification of proof rules for (object-level) connectives.

Our linear meta-logic will yield specifications of object-logic proof systems only after we assign polarities to atoms of the form $\lfloor \cdot \rfloor$ and $\lceil \cdot \rceil$: then our adequacy results will involve establishing relationships between focused meta-level proofs and object-level proof systems.

3 Sequent Calculus

We first consider how to encode sequent calculus systems for minimal, intuitionistic, and classical logics. The following three sets of formulas

$$\mathcal{L}_{lm} = (\mathcal{L} \setminus \{\bot_L, \Rightarrow_L\}) \cup \{Id_1, Id_2, Str_L, \Rightarrow'_L\} \qquad \mathcal{L}_{lj} = \mathcal{L}_{lm} \cup \{\bot_L, W_R\}$$
$$\mathcal{L}_{lk} = \mathcal{L} \cup \{Id_1, Id_2, Str_L, Str_R\}$$

where $\Rightarrow'_L$ is the formula $?\exists A \exists B[\lfloor A \Rightarrow B \rfloor^\bot \otimes (!\lceil A \rceil \otimes \lfloor B \rfloor)]$, are used to encode the LM, LJ and LK sequent calculus proof systems for minimal, intuitionistic,

$$\frac{\vdash \Theta : \Gamma \Uparrow L}{\vdash \Theta : \Gamma \Uparrow L, \bot}\ [\bot] \qquad \frac{\vdash \Theta : \Gamma \Uparrow L, F, G}{\vdash \Theta : \Gamma \Uparrow L, F \parr G}\ [\parr] \qquad \frac{\vdash \Theta, F : \Gamma \Uparrow L}{\vdash \Theta : \Gamma \Uparrow L, ?F}\ [?]$$

$$\frac{}{\vdash \Theta : \Gamma \Uparrow L, \top}\ [\top] \qquad \frac{\vdash \Theta : \Gamma \Uparrow L, F \quad \vdash \Theta : \Gamma \Uparrow L, G}{\vdash \Theta : \Gamma \Uparrow L, F \,\&\, G}\ [\&] \qquad \frac{\vdash \Theta : \Gamma \Uparrow L, F[c/x]}{\vdash \Theta : \Gamma \Uparrow L, \forall x\, F}\ [\forall]$$

$$\frac{}{\vdash \Theta : \Downarrow 1}\ [1] \qquad \frac{\vdash \Theta : \Gamma \Downarrow F \quad \vdash \Theta : \Gamma' \Downarrow G}{\vdash \Theta : \Gamma, \Gamma' \Downarrow F \otimes G}\ [\otimes] \qquad \frac{\vdash \Theta : \Uparrow F}{\vdash \Theta : \Downarrow !\, F}\ [!]$$

$$\frac{\vdash \Theta : \Gamma \Downarrow F}{\vdash \Theta : \Gamma \Downarrow F \oplus G}\ [\oplus_l] \qquad \frac{\vdash \Theta : \Gamma \Downarrow G}{\vdash \Theta : \Gamma \Downarrow F \oplus G}\ [\oplus_r] \qquad \frac{\vdash \Theta, F : \Gamma \Downarrow F[t/x]}{\vdash \Theta : \Gamma \Downarrow \exists x\, F}\ [\exists]$$

$$\frac{}{\vdash \Theta : A_p^\perp \Downarrow A_p}\ [I_1] \qquad \frac{}{\vdash \Theta, A_p^\perp : \Downarrow A_p}\ [I_2] \qquad \frac{\vdash \Theta : \Gamma, S \Uparrow L}{\vdash \Theta : \Gamma \Uparrow L, S}\ [R \Uparrow]$$

$$\frac{\vdash \Theta : \Gamma \Downarrow P}{\vdash \Theta : \Gamma, P \Uparrow}\ [D_1] \qquad \frac{\vdash \Theta, P : \Gamma \Downarrow P}{\vdash \Theta, P : \Gamma \Uparrow}\ [D_2] \qquad \frac{\vdash \Theta : \Gamma \Uparrow N}{\vdash \Theta : \Gamma \Downarrow N}\ [R \Downarrow]$$

Fig. 3. The focused proof system for linear logic [And92]. Here, L is a list of formulas, Θ is a multiset of formulas, Γ is a multiset of literals and positive formulas, A_p is a positive literal, N is a negative formula, P is not a negative literal, and S is a positive formula or a negated atom.

$$\dfrac{\dfrac{}{\vdash \mathcal{K} : \Downarrow \lfloor A \Rightarrow B \rfloor^\perp}\ [I_2] \qquad \dfrac{\dfrac{\vdash \mathcal{K} : \lceil A \rceil \Uparrow}{\vdash \mathcal{K} : \Downarrow !\lceil A \rceil}\ [!, R \Uparrow] \qquad \dfrac{\vdash \mathcal{K} : \lfloor B \rfloor, \lceil C \rceil \Uparrow}{\vdash \mathcal{K} : \lceil C \rceil \Downarrow \lfloor B \rfloor}\ [R \Downarrow, R \Uparrow]}{\vdash \mathcal{K} : \lceil C \rceil \Downarrow !\lceil A \rceil \otimes \lfloor B \rfloor}\ [\otimes]}{\dfrac{\vdash \mathcal{K} : \lceil C \rceil \Downarrow F}{\vdash \mathcal{K} : \lceil C \rceil \Uparrow \cdot}\ [D_2]}\ [2 \times \exists, \otimes]$$

Fig. 4. Here, the formula $A \Rightarrow B \in \Gamma$ and $\mathcal{K}$ denotes the set $\mathcal{L}_{lm}, \lfloor \Gamma \rfloor$

and classical logic (not displayed here to save space). These sets differ in the structural rules for $\lceil \cdot \rceil$, in the presence or absence of the formula $\bot_L$ and in the formula encoding the left introduction for implication: in the LM encoding, no structural rule is allowed in the right-hand-side formula; in the LJ encoding, the right-hand formula can be weakened; and in the LK encoding, contraction is also allowed (using the exponential ?). The $\bot_L$ formula only appears in the encodings of LJ and LK. In the theories for LM and LJ, the formula encoding the left introduction rule for implication contains a !. We will comment more about this difference later in this section.

If we fix the polarity of all meta-level atoms to be negative, then focused proofs using $\mathcal{L}_{lm}$, $\mathcal{L}_{lj}$, and $\mathcal{L}_{lk}$ yield encodings of the object-level proofs in LM, LJ, and LK. To illustrate why focusing is relevant, consider the encoding of the left introduction rule for $\Rightarrow$: selecting this rule at the object-level corresponds to focusing on the formula $F = \exists A \exists B[\lfloor A \Rightarrow B \rfloor^\perp \otimes (!\lceil A \rceil \otimes \lfloor B \rfloor)]$ (which is a member of $\mathcal{L}_{lm}$). The focused derivation in Figure 4 is then forced once F is selected for the focus: for example, the left-hand-side subproof must be an application of initial – nothing else will work with the focusing discipline. Notice that this meta-level derivation directly encodes the usual left introduction rule for $\Rightarrow$: the object-level sequents $\Gamma, B \vdash C$ and $\Gamma \vdash A$ yields $\Gamma, A \Rightarrow B \vdash C$.

Proposition 1. *Let $\Gamma \cup \Delta \cup \{C\}$ be a set of object-level formulas. Assume that all meta-level atomic formulas are given a negative polarity. Then*

1) $\Gamma \vdash_{lm} C$ *iff* $\vdash \mathcal{L}_{lm}, \lfloor \Gamma \rfloor : \lceil C \rceil \Uparrow$ 2) $\Gamma \vdash_{lj} C$ *iff* $\vdash \mathcal{L}_{lj}, \lfloor \Gamma \rfloor : \lceil C \rceil \Uparrow$
3) $\Gamma \vdash_{lk} \Delta$ *iff* $\vdash \mathcal{L}_{lk}, \lfloor \Gamma \rfloor, \lceil \Delta \rceil :\Uparrow$

This proposition is proved in [MP02, Pim01]. As stated, this proposition is a relative completeness result. It is easy to see that, for LM, LJ, and LK, we can obtain full completeness of *proofs* result: that is, focusing proofs using $\mathcal{L}_{lm}$, $\mathcal{L}_{lj}$, or $\mathcal{L}_{lk}$ correspond directly to object-level sequent calculus proofs in LM, LJ, or LK, respectively. As is apparent from the example above concerning the left-introduction rule for $\Rightarrow$, we can actually get a full completeness of *derivations* result: inference rules in the object-level sequents are in one-to-one correspondence with focused derivations in the meta-logic. To achieve this level of adequacy, the ! in the encoding of the implication left-introduction rule is important for minimal and intuitionistic logics.

If one removes the formula Id_2 from the sets $\mathcal{L}_{lm}$, $\mathcal{L}_{lj}$, and $\mathcal{L}_{lk}$, obtaining the sets $\mathcal{L}^f_{lm}$, $\mathcal{L}^f_{lj}$, and $\mathcal{L}^f_{lk}$, respectively, one can restrict the proofs encoded to cut free (object-level) proofs, represented by the judgments $\vdash^f_{lm}$ for minimal logic, $\vdash^f_{lj}$ for intuitionistic logic, and $\vdash^f_{lk}$ for classical logic.

Proposition 2. *Let $\Gamma \cup \Delta \cup \{C\}$ be a set of object-level formulas. Assume that all meta-level atomic formulas are given a negative polarity. Then*

1) $\Gamma \vdash^f_{lm} C$ *iff* $\vdash \mathcal{L}^f_{lm}, \lfloor \Gamma \rfloor : \lceil C \rceil \Uparrow$ 2) $\Gamma \vdash^f_{lj} C$ *iff* $\vdash \mathcal{L}^f_{lj}, \lfloor \Gamma \rfloor : \lceil C \rceil \Uparrow$
3) $\Gamma \vdash^f_{lk} \Delta$ *iff* $\vdash \mathcal{L}^f_{lk}, \lfloor \Gamma \rfloor, \lceil \Delta \rceil :\Uparrow$

As above, similar results of full completeness of both proofs and derivations can be proved.

4 Natural Deduction

The system depicted in Figure 5 is a intuitionistic variant of the classical system in [SB98], presenting natural deduction using a sequent-style notation: sequents of the form $\Gamma \vdash_{nd} C \uparrow$, encoded as a meta-level sequent $\vdash \Sigma, \lfloor \Gamma \rfloor : \lceil C \rceil$ (for some multiset of formulas Σ), are obtained from the conclusion by a derivation (from bottom-up) where C is not the major premise of an elimination rule; and sequents of the form $\Gamma \vdash_{nd} C \downarrow$, encoded as a sequent $\vdash \Sigma, \lfloor \Gamma \rfloor : \lfloor C \rfloor^{\perp}$, are obtained from the set of hypotheses by a derivation (from top-down) where C is extracted from the major premise of an elimination rule. These two types of derivations meet either with a match rule M or with a switch rule S. These two types of sequents are used to distinguish general natural deduction proofs from the normal form proofs, where the switch rule is not allowed. More precisely, normal proofs here

$$\frac{}{\Gamma, A \vdash_{nd} A \downarrow}\,[\mathrm{Ax}] \quad \frac{\Gamma \vdash_{nd} F \uparrow \quad \Gamma \vdash_{nd} G \uparrow}{\Gamma \vdash_{nd} F \wedge G \uparrow}\,[\wedge I] \quad \frac{\Gamma \vdash_{nd} F \wedge G \downarrow}{\Gamma \vdash_{nd} F \downarrow}\,[\wedge E]$$

$$\frac{\Gamma \vdash_{nd} A_i \uparrow}{\Gamma \vdash_{nd} A_1 \vee A_2 \uparrow}\,[\vee I] \quad \frac{\Gamma \vdash_{nd} A \vee B \downarrow \;\; \Gamma, A \vdash_{nd} C \uparrow (\downarrow) \; \Gamma, B \vdash_{nd} C \uparrow (\downarrow)}{\Gamma \vdash_{nd} C \uparrow (\downarrow)}\,[\vee E]$$

$$\frac{\Gamma, A \vdash_{nd} B \uparrow}{\Gamma \vdash_{nd} A \Rightarrow B \uparrow}\,[\Rightarrow I] \quad \frac{\Gamma \vdash_{nd} A \Rightarrow B \downarrow \quad \Gamma \vdash_{nd} A \uparrow}{\Gamma \vdash_{nd} B \downarrow}\,[\Rightarrow E] \quad \frac{}{\Gamma \vdash_{nd} t \uparrow}\,[tI]$$

$$\frac{\Gamma \vdash_{nd} A\{c/x\} \uparrow}{\Gamma \vdash_{nd} \forall x\, A \uparrow}\,[\forall I] \quad \frac{\Gamma \vdash_{nd} \forall x\, A \downarrow}{\Gamma \vdash_{nd} A\{t/x\} \downarrow}\,[\forall E] \quad \frac{\Gamma \vdash_{nd} A \downarrow}{\Gamma \vdash_{nd} A \uparrow}\,[\mathrm{M}] \quad \frac{\Gamma \vdash_{nd} A \uparrow}{\Gamma \vdash_{nd} A \downarrow}\,[\mathrm{S}]$$

$$\frac{\Gamma \vdash_{nd} \exists x\, A \downarrow \quad \Gamma, A\{a/x\} \vdash_{nd} C \uparrow (\downarrow)}{\Gamma \vdash_{nd} C \uparrow (\downarrow)}\,[\exists E] \quad \frac{\Gamma \vdash_{nd} A\{t/x\} \uparrow}{\Gamma \vdash_{nd} \exists x\, A \uparrow}\,[\exists I]$$

Fig. 5. Rules for minimal natural deduction - NM. In $[\vee L]$, $i \in \{1, 2\}$.

coincide with the normal proofs as in [Pra65] only in the $\forall, \wedge$ and $\Rightarrow$ fragment. We use the judgment $\vdash_{nd}$ to denote the existence of a natural deduction proof and the judgment $\vdash_{nd}^{n}$ to denote the existence of a normal natural deduction proof.

We can account for natural deduction in minimal logic by simply changing polarity assignment: in particular, atoms of the form $\lfloor \cdot \rfloor$ are now positive and all atoms of the form $\lceil \cdot \rceil$ have negative polarity. This change in polarity causes the formula Id_2, which behaved like the cut rule in sequent calculus, to now behave like the switch rule, as illustrated by the following derivation.

$$\cfrac{\cfrac{\cfrac{}{\vdash \Sigma, \lfloor \Gamma \rfloor : \lfloor C \rfloor^{\perp} \Downarrow \lfloor C \rfloor}\,[I_1] \quad \cfrac{\vdash \Sigma, \lfloor \Gamma \rfloor : \lceil C \rceil \Uparrow}{\vdash \Sigma, \lfloor \Gamma \rfloor : \Downarrow \lceil C \rceil}\,[R\Downarrow, R\Uparrow]}{\vdash \Sigma, \lfloor \Gamma \rfloor : \lfloor C \rfloor^{\perp} \Downarrow \lfloor C \rfloor \otimes \lceil C \rceil}\,[\otimes]}{\vdash \Sigma, \lfloor \Gamma \rfloor : \lfloor C \rfloor^{\perp} \Uparrow}\,[D_2, \exists]$$

As the following proposition states, to obtain an encoding of normal form proofs, we do not include the formula Id_2.

Proposition 3. *Let $\Gamma \cup \{C\}$ be a set of object-level formulas and assume that all $\lceil \cdot \rceil$ atomic formulas are given a negative polarity and that all $\lfloor \cdot \rfloor$ atomic formulas are given a positive polarity. Then*

1) $\Gamma \vdash_{nm} C\uparrow$ *iff* $\vdash \mathcal{L}_{lm}, \lfloor \Gamma \rfloor : \lceil C \rceil \Uparrow$ 2) $\Gamma \vdash_{nm}^{n} C\uparrow$ *iff* $\vdash \mathcal{L}_{lm}^{f}, \lfloor \Gamma \rfloor : \lceil C \rceil \Uparrow$
3) $\Gamma \vdash_{nm}^{n} C\downarrow$ *iff* $\vdash \mathcal{L}_{lm}^{f}, \lfloor \Gamma \rfloor : \lfloor C \rfloor^{\perp} \Uparrow$

An equivalent full completeness of proofs statement can also be proved.

Since the polarity assignment in a focused system does not affect provability, we obtain for free the following relative completeness result between LM and NM.

Corollary 1. *If $\Gamma \cup \{C\}$ be a set of object-level formulas, then*

$$\Gamma \vdash_{lm} C \textit{ iff } \Gamma \vdash_{nm} C \quad \textit{and} \quad \Gamma \vdash_{lm}^{f} C \textit{ iff } \Gamma \vdash_{nm}^{n} C.$$

Treating negation (in particular, falsity) in natural deduction presentations of intuitionistic and classical logics is not straightforward. We show in [NM08] that extra meta-logic formulas are needed to encode these systems. Since the treatment of negation in natural deduction is not one about focusing in the meta-level, we do not discuss this issue further here.

5 Natural Deduction with General Elimination Rules

Schroeder-Heister proposed an extension of natural deduction in [SH84], which we call "general natural deduction", by using the general elimination rules, depicted in Figure 6, that treats all elimination rules in the same indirect style that is usually used for disjunction elimination rule. To encode proofs in the general natural deduction, we assign negative polarity to $\lfloor\cdot\rfloor$ and $\lceil\cdot\rceil$ atoms, and use the set of formulas $\mathcal{L}_{ge}$, obtained from $\mathcal{L}_{lm}$ by removing the formulas $\vee_L, \wedge_L, \Rightarrow'_L, \forall_L$ and adding the existential closure of the following four formulas:

$$\begin{array}{ll} \lceil A \Rightarrow B\rceil \otimes (!\lceil A\rceil \otimes \lfloor B\rfloor) & \lceil \forall B\rceil \otimes \lfloor Bx\rfloor \\ \lceil A \vee B\rceil \otimes (\lfloor A\rfloor \mathbin{\&} \lfloor B\rfloor) & \lceil A \wedge B\rceil \otimes (\lfloor A\rfloor \mathbin{⅋} \lfloor B\rfloor) \end{array}$$

Proposition 4. *Let $\Gamma \cup \{C\}$ be a set of object-level formulas. Assume that all meta-level atomic formulas are given a negative polarity. Then $\Gamma \vdash_{ge} C$ iff $\vdash \mathcal{L}_{ge}, \lfloor\Gamma\rfloor : \lceil C\rceil \Uparrow$.*

An equivalent full completeness of proofs statement can also be proved.

Notice that there are two differences between the formulas displayed above and the original formulas in $\mathcal{L}_{lm}$ that they replace. 1) The presence of the multiplicative version of $\wedge_L$, and 2) the replacement of literals of the form $\lfloor B\rfloor^\perp$ by $\lceil B\rceil$. Moreover, notice that without the Id_2 formula the equivalence $\lfloor B\rfloor^\perp \equiv \lceil B\rceil$ is not satisfied and, therefore, the set of formulas in $\mathcal{L}_{ge}$ is not equivalent to $\mathcal{L}^f_{lm}$. Therefore, we relate general natural deduction to the formulation of LM that contains the cut rule.

Corollary 2. *Let $\Gamma \cup \{C\}$ be a set of object-level formulas. Then $\Gamma \vdash_{ge} C$ iff $\Gamma \vdash_{lm} C$.*

Negri and Plato in [NP01] propose a different notion of normal proofs in general natural deduction: *Derivations in general normal form have all major premises of elimination rules as assumption.* In other words, the major premises, represented by the bracketed formula in the general elimination rules shown in

$$\frac{\Gamma \vdash_{ge} [A \vee B] \quad \Gamma, A \vdash_{ge} C \quad \Gamma, B \vdash_{ge} C}{\Gamma \vdash_{ge} C} \qquad \frac{\Gamma \vdash_{ge} [A \wedge B] \quad \Gamma, A, B \vdash_{ge} C}{\Gamma \vdash_{ge} C}$$

$$\frac{\Gamma \vdash_{ge} [A \Rightarrow B] \quad \Gamma \vdash_{ge} A \quad \Gamma, B \vdash_{ge} C}{\Gamma \vdash_{ge} C} \qquad \frac{\Gamma \vdash_{ge} [\forall x\, A] \quad \Gamma, A\{t/x\} \vdash_{ge} C}{\Gamma \vdash_{ge} C}$$

Fig. 6. Four general elimination rules. The major premise is marked with brackets.

Figure 6, are discharged assumptions. In our framework, this amounts to enforcing, by the use of polarity assignment to meta-level atoms, that the major premises are present in the set of assumptions. We use the set $\mathcal{L}^f_{lm}$ and assign negative polarity to all atoms of the form $\lfloor\cdot\rfloor$ and $\lceil\cdot\rceil$, to encode general normal form proofs, represented by the judgment $\vdash^n_{ge}$.

Proposition 5. *Let $\Gamma \cup \{C\}$ be a set of object-level formulas. Assume that all meta-level atomic formulas are given a negative polarity. Then $\Gamma \vdash^n_{ge} C$ iff $\vdash \mathcal{L}^f_{lm}, \lfloor\Gamma\rfloor : \lceil C\rceil \Uparrow$.*

An equivalent full completeness of proofs statement can also be proved.

It is easy to see in our framework that cut-free sequent calculus proofs can easily be obtained from general normal forms proofs, and vice-versa, since, to encode both systems, we use exactly the same formulas, $\mathcal{L}^f_{lm}$, and assign the same polarity to $\lfloor\cdot\rfloor$ and $\lceil\cdot\rceil$ atoms.

Corollary 3. *Let Γ be a set of formulas and let C be a formula. Then $\Gamma \vdash^n_{ge} C$ iff $\Gamma \vdash^f_{lm} C$.*

6 Free Deduction

In [Par92], Parigot introduced the *free deduction* proof system for classical logic that employed both the general elimination rules of the previous section and general introduction rules[1]. The general introduction rules are depicted in Figure 7.

$$\frac{\Gamma, A \vee B \vdash_{fd} \Delta \quad \Gamma \vdash_{fd} \Delta, A}{\Gamma \vdash_{fd} \Delta}\ [\vee GI] \qquad \frac{\Gamma, A \Rightarrow B \vdash_{fd} \Delta \quad \Gamma, A \vdash_{fd} \Delta, B}{\Gamma \vdash_{fd} \Delta}\ [\Rightarrow GI]$$

$$\frac{\Gamma, A \wedge B \vdash_{fd} \Delta \quad \Gamma \vdash_{fd} \Delta, A \quad \Gamma \vdash_{fd} \Delta, B}{\Gamma \vdash_{fd} \Delta}\ [\wedge GI]$$

$$\frac{\Gamma, \neg A \vdash_{fd} \Delta \quad \Gamma, A \vdash_{fd} \Delta}{\Gamma \vdash_{fd} \Delta}\ [\neg GI_1] \qquad \frac{\Gamma \vdash_{fd} \Delta, \neg A \quad \Gamma \vdash_{fd} \Delta, A}{\Gamma \vdash_{fd} \Delta}\ [\neg GI_2]$$

Fig. 7. The general introduction rules

To encode free deduction proofs, we proceed similarly to the treatment of natural deduction with general eliminations rules. In particular, we replace in all formulas of $\mathcal{L}$, except the formula $\bot_L$, literals of the form $\lfloor B\rfloor^\perp$ by $\lceil B\rceil$ and literals of the form $\lceil B\rceil^\perp$ by $\lfloor B\rfloor$, and call the resulting set union $\{Id_1, Id_2, Str_L, Str_R\}$ as $\mathcal{L}_{fd}$. For example, the formula $\wedge_R$ in $\mathcal{L}$ is replaced by $?\exists A\exists B[\lfloor A \wedge B\rfloor \otimes (\lceil A\rceil \,\&\, \lceil B\rceil)]$ in $\mathcal{L}_{fd}$.

We assign negative polarity to the atoms $\lfloor\cdot\rfloor$ and $\lceil\cdot\rceil$ except the atom $\lfloor\bot\rfloor$, for which we assign positive polarity because of the different treatment of negation in free deduction.

[1] It is interesting to note that later and independently, Negri and Plato also introduced general introduction rules in [NP01, p. 214].

Proposition 6. *Let $\Gamma \cup \Delta$ be a set of object-level formulas. Assume that all meta-level atomic formulas are given a negative polarity except the atom $\lfloor \bot \rfloor$, which is given positive polarity. Then $\Gamma \vdash_{fd} \Delta$ iff $\vdash \mathcal{L}_{fd}, \lfloor \Gamma \rfloor, \lceil \Delta \rceil :\Uparrow$.*

Full completeness for both proofs and derivations can also be proved.

Since the encoding $\mathcal{L}_{fd}$ is logically equivalent to $\mathcal{L}_{lk}$, we can show that free deduction and LK are relative complete.

Corollary 4. *Let Γ and Δ be sets of formulas. Then $\Gamma \vdash_{fd} \Delta$ iff $\Gamma \vdash_{lk} \Delta$.*

Parigot notes that if one of the premises of the general rules is “killed”, *i.e.*, it is always the conclusion of an initial rule, then one can obtain either sequent calculus or natural deduction proofs. The “killing” of a premise is accounted for in our framework by the use of polarities to enforce the presence of a formula in the context of the sequent. As done with the normal forms in general natural deduction, we can use the equivalences $\lfloor B \rfloor \equiv \lceil B \rceil^\perp$ and $\lfloor B \rfloor^\perp \equiv \lceil B \rceil$ and use either additive or multiplicative versions of the formulas in $\mathcal{L}$ to obtain from $\mathcal{L}_{fd}$ the equivalent sets $\mathcal{L}_{lk}$, which encodes LK, and the set $\mathcal{L}_{fd}^{nk}$ obtained from $\mathcal{L}_{lk}$ by removing the formulas $\Rightarrow_L, \vee_L, \wedge_L$ and adding the existential closure of the following three clauses:

$$\begin{array}{ll} \lceil A \Rightarrow B \rceil \otimes \left(\lceil A \rceil \otimes \lceil B \rceil^\perp\right) & \lceil A \wedge B \rceil \otimes \left(\lceil A \rceil^\perp \oplus \lceil B \rceil^\perp\right) \\ \lceil A \vee B \rceil \otimes \left(\lceil A \rceil^\perp \otimes \lceil B \rceil^\perp\right). & \end{array}$$

The resulting set of formulas can be seen as an encoding of a multiple conclusion natural deduction proof system.

7 System KE

In the previous sections, we dealt with systems that contained rules with more premises than the corresponding rules in sequent calculus or natural deduction. Now, we move to the other direction and deal with systems that contain rules with fewer premises.

In [DM94], D’Agostino and Mondadori proposed the propositional tableaux system KE displayed in Figure 8. Here, the only rule that has more than one premise is the cut rule. In the original system, the cut inference rule appears with a side condition limiting cuts to be analytical cuts. Though it is possible to encode analytic cuts in our framework, as we show in [NM08], we consider here the more general form of cuts because it relates more directly to the other systems already presented.

To encode KE, we assign negative polarity to all atoms $\lfloor \cdot \rfloor$ and $\lceil \cdot \rceil$ and use the set of linear logic formulas, $\mathcal{L}_{ke}$, obtained from $\mathcal{L}_{lk}^p$ (the propositional fragment of $\mathcal{L}_{lk}$), by removing the formulas $\wedge_R, \Rightarrow_L, \vee_L, \vee_R, \bot_L$ and adding the existential closure of the following eight formulas:

$$\begin{array}{ll} \lfloor A \Rightarrow B \rfloor^\perp \otimes (\lfloor A \rfloor^\perp \otimes \lfloor B \rfloor) & \lceil A \wedge B \rceil^\perp \otimes (\lfloor A \rfloor^\perp \otimes \lceil B \rceil) \\ \lfloor A \Rightarrow B \rfloor^\perp \otimes (\lceil A \rceil \otimes \lceil B \rceil^\perp) & \lceil A \wedge B \rceil^\perp \otimes (\lceil A \rceil \otimes \lfloor B \rfloor^\perp) \\ \lfloor A \vee B \rfloor^\perp \otimes (\lceil A \rceil^\perp \otimes \lfloor B \rfloor) & \lceil A \vee B \rceil^\perp \otimes (\lceil A \rceil \parr \lceil B \rceil) \\ \lfloor A \vee B \rfloor^\perp \otimes (\lfloor A \rfloor \otimes \lceil B \rceil^\perp) & \lceil \bot \rceil \end{array}$$

$$\frac{\Gamma, A \vee B, B \vdash_{ke} A, \Delta}{\Gamma, A \vee B \vdash_{ke} A, \Delta}\ [\vee_{L1}] \quad \frac{\Gamma, A \vee B, A \vdash_{ke} B, \Delta}{\Gamma, A \vee B \vdash_{ke} B, \Delta}\ [\vee_{L2}] \quad \frac{\Gamma \vdash_{ke} A, B, A \vee B, \Delta}{\Gamma \vdash_{ke} A \vee B, \Delta}\ [\vee_R]$$

$$\frac{\Gamma, A \wedge B, A, B \vdash_{ke} \Delta}{\Gamma, A \wedge B \vdash_{ke} \Delta}\ [\wedge_L] \quad \frac{\Gamma, A \vdash_{ke} A \wedge B, B, \Delta}{\Gamma, A \vdash_{ke} A \wedge B, \Delta}\ [\wedge_{R1}] \quad \frac{\Gamma, B \vdash_{ke} A \wedge B, A, \Delta}{\Gamma, B \vdash_{ke} A \wedge B, \Delta}\ [\wedge_{R1}]$$

$$\frac{\Gamma, A, A \Rightarrow B, B \vdash_{ke} \Delta}{\Gamma, A, A \Rightarrow B \vdash_{ke} \Delta}\ [\Rightarrow_{L1}] \quad \frac{\Gamma, A \Rightarrow B \vdash_{ke} A, B, \Delta}{\Gamma, A \Rightarrow B \vdash_{ke} B, \Delta}\ [\Rightarrow_{L2}]$$

$$\frac{\Gamma, \neg A \vdash_{ke} A, \Delta}{\Gamma, \neg A \vdash_{ke} \Delta}\ [\neg_L] \quad \frac{\Gamma, A \vdash_{ke} \neg A, \Delta}{\Gamma \vdash_{ke} \neg A, \Delta}\ [\neg_R] \qquad \frac{\Gamma, A \vdash_{ke} A \Rightarrow B, B, \Delta}{\Gamma \vdash_{ke} A \Rightarrow B, \Delta}\ [\Rightarrow_R]$$

$$\frac{}{\Gamma, A \vdash_{ke} A, \Delta}\ [Ax] \quad \frac{\Gamma, A \vdash_{ke} \Delta \quad \Gamma \vdash_{ke} A, \Delta}{\Gamma \vdash_{ke} \Delta}\ [Cut]$$

Fig. 8. The rules for the classical propositional logic KE

Proposition 7. *Let $\Gamma \cup \Delta$ be a set of object-level formulas. Assume that all meta-level atomic formulas are given a negative polarity. Then $\Gamma \vdash_{ke} \Delta$ iff $\vdash \mathcal{L}_{ke}, \lfloor \Gamma \rfloor, \lceil \Delta \rceil :\Uparrow$.*

Full completeness of both proofs and derivations can also be proved.

The only differences between $\mathcal{L}_{lk}^{p}$ and $\mathcal{L}_{ke}$ are the use of multiplicative connectives instead of additive connectives, and that some atoms of the form $\lfloor\cdot\rfloor$ ($\lceil\cdot\rceil$) appear in the form $\lceil\cdot\rceil^{\perp}$ ($\lfloor\cdot\rfloor^{\perp}$). As before, we can show that the sets $\mathcal{L}_{lk}^{p}$ and $\mathcal{L}_{ke}$ are equivalent: the first difference is addressed by the presence of Str_L and Str_R and the second difference is addressed by the presence of Id_1 and Id_2.

Corollary 5. *Let Γ and Δ be a set of formulas. Then $\Gamma \vdash_{ke} \Delta$ iff $\Gamma \vdash_{lk}^{p} \Delta$, where $\vdash_{lk}^{p}$ is the judgment representing provability in the propositional fragment of LK.*

8 Smullyan's Analytic Cut System

To illustrate how one can capture another extreme in proof systems, we consider Smullyan's proof system for analytic cut (AC) [Smu68], which is depicted in Figure 9. Here, all rules except the cut rule are axioms. As the name of the system suggests, Smullyan also assigned a side condition to the cut rule, allowing only analytical cuts. As in the previous section, we shall drop this restriction in order to make connections to previous systems easier (but we can account for it: see [NM08]).

We again assign negative polarity to $\lfloor\cdot\rfloor$ and $\lceil\cdot\rceil$ atoms and use the theory $\mathcal{L}_{ac}$ that results from collecting the formulas in $\{Id_1, Id_2, Str_L, Str_L\}$ with the formula $\lceil\perp\rceil$ and the existential closure of the following formulas:

$$\frac{}{\Gamma, A \vee B \vdash_{ac} A, B, \Delta}\ [\vee_L] \qquad \frac{}{\Gamma, A \vdash_{ac} A \vee B, \Delta}\ [\vee_{R1}] \qquad \frac{}{\Gamma, B \vdash_{ac} A \vee B, \Delta}\ [\vee_{R2}]$$

$$\frac{}{\Gamma, A \wedge B \vdash_{ac} A, \Delta}\ [\wedge_{L1}] \qquad \frac{}{\Gamma, A \wedge B \vdash_{ac} B, \Delta}\ [\wedge_{L2}] \qquad \frac{}{\Gamma, A, B \vdash_{ac} A \wedge B, \Delta}\ [\wedge_R]$$

$$\frac{}{\Gamma, A, A \Rightarrow B \vdash_{ac} B, \Delta}\ [\Rightarrow_L] \qquad \frac{}{\Gamma \vdash_{ac} A, A \Rightarrow B, \Delta}\ [\Rightarrow_{R1}] \qquad \frac{}{\Gamma, B \vdash_{ac} A \Rightarrow B, \Delta}\ [\Rightarrow_{R2}]$$

$$\frac{}{\Gamma, \neg A, A \vdash_{ac} \Delta}\ [\neg_L] \qquad \frac{}{\Gamma \vdash_{ac} A, \neg A, \Delta}\ [\neg_R] \qquad \frac{}{\Gamma, A \vdash_{ac} A, \Delta}\ [Ax]$$

$$\frac{\Gamma, A \vdash_{ac} \Delta \quad \Gamma \vdash_{ac} A, \Delta}{\Gamma \vdash_{ac} \Delta}\ [Cut]$$

Fig. 9. Smullyan's Analytic Cut System AC for classical propositional logic, except that the cut rule is not restricted

$$\begin{array}{ll}
\lfloor A \wedge B \rfloor^{\perp} \otimes \left(\lceil A \rceil^{\perp} \oplus \lceil B \rceil^{\perp}\right) & \lceil A \wedge B \rceil^{\perp} \otimes \left(\lfloor A \rfloor^{\perp} \otimes \lfloor B \rfloor^{\perp}\right) \\
\lfloor A \vee B \rfloor^{\perp} \otimes \left(\lceil A \rceil^{\perp} \otimes \lceil B \rceil^{\perp}\right) & \lceil A \vee B \rceil^{\perp} \otimes \left(\lfloor A \rfloor^{\perp} \oplus \lfloor B \rfloor^{\perp}\right) \\
\lfloor A \Rightarrow B \rfloor^{\perp} \otimes \left(\lfloor A \rfloor^{\perp} \otimes \lceil B \rceil^{\perp}\right) & \lceil A \Rightarrow B \rceil^{\perp} \otimes \left(\lceil A \rceil^{\perp} \oplus \lfloor B \rfloor^{\perp}\right)
\end{array}$$

Proposition 8. *Let $\Gamma \cup \Delta$ be a set of object-level formulas. Assume that all meta-level atomic formulas are given a negative polarity. Then $\Gamma \vdash_{ac} \Delta$ iff $\vdash \mathcal{L}_{ac}, \lfloor \Gamma \rfloor, \lceil \Delta \rceil :\Uparrow$.*

Equivalent results of full completeness of both proofs and derivations can be proved.

The encoding above differs from $\mathcal{L}_{lk}^{p}$ as in ways similar to the differences between $\mathcal{L}_{lk}^{p}$ and $\mathcal{L}_{ke}$. By using the same reasoning as with the encoding $\mathcal{L}_{ke}$, we can show that AC is (Level 0) equivalent to the propositional fragment of LK.

Corollary 6. *Let Γ and Δ be a set of formulas. Then $\Gamma \vdash_{ac} \Delta$ iff $\Gamma \vdash_{lk}^{p} \Delta$, where $\vdash_{lk}^{p}$ is the judgment representing provability in the propositional fragment of LK.*

9 Related Work

A number of logical frameworks have been proposed to represent object-level proof systems. Many of these frameworks, as used in [FM88, HHP93, Pfe89], are based on intuitionistic (minimal) logic principles. In such settings, the dualities that we employ here, for example, $\lfloor B \rfloor \equiv \lceil B \rceil^{\perp}$, are not available within the logic and this makes reasoning about the relative completeness between object-level proof systems harder. Also, since minimal logic sequents must have a single conclusion, the storage of object-level formulas is generally done on the left-hand side of meta-level sequents (see [HM94, Pfe00]) with some kind of "marker" for the right-hand side (such as the non-logical "refutation" marker # in [Pfe00]). The flexibility of having the four meta-level literals $\lfloor B \rfloor$, $\lceil B \rceil$, $\lfloor B \rfloor^{\perp}$, and $\lceil B \rceil^{\perp}$ is not generally available in such intuitionistic systems. While it is natural in classical linear logic to consider having some atoms assigned negative and some positive polarities, most intuitionistic systems consider only uniform assignments of polarities to meta-level atoms (usually negative in order to support goal-directed proof search): the ability to mix polarity assignments for different meta-level atoms can only be achieved in more indirect fashions in such settings.

The abstract logic programming presentation of linear logic called Forum [Mil96] has been used to specify sequent calculus proof systems in a style similar to that used here. That presentation of linear logic was, however, also limited in that negation was not a primitive connective and that all atomic formulas were assumed to have negative polarity. The range of encodings contained in this paper are not directly available using Forum.

10 Conclusions and Further Remarks

We have shown that by employing different focusing annotations or using different sets of formulas that are (meta-logically) equivalent to $\mathcal{L}$, a range of sound and (relatively) complete object-level proof systems could be encoded. We have illustrated this principle by showing how linear logic focusing and logical equivalences can account for object-level proof systems based on sequent calculus, natural deduction, generalized introduction and elimination rules, free deduction, the tableaux system KE, and Smullyan's system employing only axioms and the cut rule.

Logical frameworks aim at allowing proof systems to be specified using compact and declarative specifications of inference rules. It now seems that a much broader range of possible proof systems can be further specified by allowing flexible assignment of polarity to meta-logical atoms (instead of making the usual assignment of some fixed, global polarity assignment). A natural next step would be to see what insights might be carried from this setting of linear-intuitionistic-classical logic to other, say, intermediate or sub-structural logics.

While focusing at the meta-level clearly provides a powerful normal form of proof, we have not described how to use the techniques presented in this paper to derive object-level focusing proof systems. Finding a means to derive such object-level normal form proofs is an interesting challenge that we plan to develop next.

Another interesting line of future research would be to consider differences in the *sizes* of proofs in these different paradigms since these differences can be related to the topic of comparing bottom-up and top-down deduction. Thus, it might be possible to flexibly change polarity assignments that would result in different and, hopefully, more compact presentations of proofs.

Acknowledgments. This work has been supported in part by INRIA through the "Equipes Associées" Slimmer and by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project.

References

[And92] Andreoli, J.-M.: Logic programming with focusing proofs in linear logic. J. of Logic and Computation 2(3), 297–347 (1992)

[DM94] D'Agostino, M., Mondadori, M.: The taming of the cut. Classical refutations with analytic cut 4(3), 285–319 (1994)

[FM88] Felty, A., Miller, D.: Specifying theorem provers in a higher-order logic programming language. In: Ninth International Conference on Automated Deduction, Argonne, IL, May 1988, pp. 61–80. Springer, Heidelberg (1988)

[Gen69] Gentzen, G.: Investigations into logical deductions. In: Szabo, M.E. (ed.) The Collected Papers of Gerhard Gentzen, pp. 68–131. North-Holland, Amsterdam (1969)

[Gir06] Girard, J.-Y.: Le Point Aveugle: Cours de logique: Tome 1, Vers la perfection. Hermann (2006)

[HHP93] Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. Journal of the ACM 40(1), 143–184 (1993)

[HM94] Hodas, J., Miller, D.: Logic programming in a fragment of intuitionistic linear logic. Information and Computation 110(2), 327–365 (1994)

[LM07] 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)

[Mil96] Miller, D.: Forum: A multiple-conclusion specification logic. Theoretical Computer Science 165(1), 201–232 (1996)

[MN07] Miller, D., Nigam, V.: Incorporating tables into proofs. In: Duparc, J., Henzinger, T.A. (eds.) CSL 2007. LNCS, vol. 4646, pp. 466–480. Springer, Heidelberg (2007)

[MP02] Miller, D., Pimentel, E.: Using linear logic to reason about sequent systems. In: Egly, U., Fermüller, C. (eds.) TABLEAUX 2002. LNCS (LNAI), vol. 2381, pp. 2–23. Springer, Heidelberg (2002)

[MP04] Miller, D., Pimentel, E.: Linear logic as a framework for specifying sequent calculus. In: van Eijck, J., van Oostrom, V., Visser, A. (eds.) Logic Colloquium 1999: Proceedings of the Annual European Summer Meeting of the Association for Symbolic Logic, pp. 111–135. A. K. Peters Ltd (2004)

[NM08] Nigam, V., Miller, D.: Focusing in linear meta-logic: Extended report, http://hal.inria.fr/inria-00281631

[NP01] Negri, S., Von Plato, J.: Structural Proof Theory. Cambridge University Press, Cambridge (2001)

[Par92] Parigot, M.: Free deduction: An analysis of "computations" in classical logic. In: Proceedings of the First Russian Conference on Logic Programming, London, UK, pp. 361–380. Springer, Heidelberg (1992)

[Pfe89] Pfenning, F.: Elf: A language for logic definition and verified metaprogramming. In: Fourth Annual Symposium on Logic in Computer Science, Monterey, CA, June 1989, pp. 313–321 (1989)

[Pfe00] Pfenning, F.: Structural cut elimination I. intuitionistic and classical logic 157(1/2), 84–141 (2000)

[Pim01] Pimentel, E.G.: Lógica linear e a especificação de sistemas computacionais. PhD thesis, Universidade Federal de Minas Gerais, Belo Horizonte, M.G., Brasil, Written in English (December 2001)

[PM05] Pimentel, E., Miller, D.: On the specification of sequent systems. In: Sutcliffe, G., Voronkov, A. (eds.) LPAR 2005. LNCS (LNAI), vol. 3835, pp. 352–366. Springer, Heidelberg (2005)

[Pra65] Prawitz, D.: Natural Deduction. Almqvist & Wiksell, Uppsala (1965)
[SB98] Sieg, W., Byrnes, J.: Normal natural deduction proofs (in classical logic). Studia Logica 60(1), 67–106 (1998)
[SH84] Schroeder-Heister, P.: A natural extension of natural deduction. Journal of Symbolic Logic 49(4), 1284–1300 (1984)
[Smu68] Smullyan, R.M.: Analytic cut. J. of Symbolic Logic 33(4), 560–564 (1968)
[vP01] von Plato, J.: Natural deduction with general elimination rules. Archive for Mathematical Logic 40(7), 541–567 (2001)

Certifying a Tree Automata Completion Checker

Benoît Boyer, Thomas Genet, and Thomas Jensen

IRISA / Université de Rennes 1 / CNRS
Campus de Beaulieu
F-35042 Rennes Cedex
{bboyer,genet,jensen}@irisa.fr

Abstract. Tree automata completion is a technique for the verification of infinite state systems. It has already been used for the verification of cryptographic protocols and the prototyping of Java static analyzers. However, as for many other verification techniques, the correctness of the associated tool becomes more and more difficult to guarantee. It is due to the size of the implementation that constantly grows and due to optimizations which are necessary to scale up the efficiency of the tool to verify real-size systems. In this paper, we define and develop a checker for tree automata produced by completion. The checker is defined using Coq and its implementation is automatically extracted from its formal specification. Using extraction gives a checker that can be run independently of the Coq environment. A specific algorithm for tree automata inclusion checking has been defined so as to avoid the exponential blow up. The obtained checker is certified in Coq, independent of the implementation of completion, usable with any approximation performed during completion, small and fast. Some benchmarks are given to show how efficient the tool is.

1 Introduction

Static program analysis is one of the cornerstones of software verification and is increasingly used to protect computing devices from malicious or mal-functioning code. However, program verifiers are themselves complex programs and a single error may jeopardize the entire trust chain of which they form part. Efforts have been made to certify static analyzers [KN03, BD04, CJPR05] or to certify the results obtained by static analyzers [LT00, BJP06] in Coq in order to increase confidence in the analyzers. In this paper, we instantiate the general framework used in [BJP06] to the particular case of analyzing term rewriting systems by tree automata completion [Gen98, FGVTT04]. Given a term rewriting system, the tree automata completion is a technique for over-approximating the set of terms reachable by rewriting in order to prove the unreachability of certain "bad" states that violate a given security property. This technique has already been used to prove security properties on cryptographic protocols [GK00], [GTTVTT03, BHK04, ABB+05, ZD06] and, more recently, to prototype static analyzers on Java byte code [BGJL07].

In this paper, we show how to mechanize the proof, within the Coq proof assistant, that the tree automaton produced by completion recognizes an over-approximation of all reachable terms. Coq is based on constructive logic (Calculus of Inductive Constructions) and it is possible to extract an Ocaml or Haskell function implementing exactly the algorithm whose specification has been expressed in Coq. The extracted code is thus a *certified* implementation of the specification given in the Coq formalism. Extracted programs are standalone and do not require the Coq environment to be executed. For details about the extraction mechanisms, readers can refer to [BC04].

A specific challenge in the work reported here has been how to marry constructive logic and efficiency. Previous case studies with tree automata completion, on cryptographic protocols [GTTVTT03] and on Java bytecode [BGJL07] show that we need an efficient completion algorithm to verify properties of real models. For instance, the current implementation of completion (called Timbuk [GVTT00]) is based on imperative data structures like hash tables whereas Coq allows only pure functional structures. A second problem is the termination of completion. Since Coq can only deal with total functions, functions must be proved terminating for any computation. In general, such a property cannot be guaranteed on completion because it mainly depends on term rewriting system and approximation equations given initially.

For these two reasons, there is little hope to specify and certify an efficient and purely functional version of the completion algorithm. Instead, we have adopted a solution based on a result-checking approach. It consists of building a smaller program (called the *checker*) - certified in Coq - that checks if the tree automaton computed by Timbuk is sound. In this paper, we restrict to the case of left-linear term rewriting systems which revealed to be sufficient for verifying Java programs [BGJL07]. However, a checker dealing with general term rewriting systems like completion does in [FGVTT04] is under development.

The closest work to ours is the one done by X. Rival and J. Goubault-Larrecq [RGL01]. They have designed a library to manipulate tree automata in Coq and proposed some optimized formal data structures that we reuse. However, we aim at dealing with larger tree automata than those used in their benchmarks. Moreover, we need some other tools which are not provided by the library as for example a specific algorithm to check inclusion.

This paper is organized as follows. Rewriting and tree automata are reviewed in Section 2 and tree automata completion in Section 3. Section 4 states the main functions to define, inclusion and closure test, and the corresponding theorems to prove. Section 5 and Section 6 give the Coq formalization of rewriting and of tree automata, respectively. The core of the checker consists of two algorithms: an optimized automata inclusion test, defined in Section 7, and a procedure for checking that an automaton is *closed* under rewriting w.r.t. a given term rewriting system, defined in Section 8. Section 9 gives some details about the performances of the checker in practice. Finally, we conclude and list some ongoing research on this subject.

2 Preliminaries

Comprehensive surveys can be found in [BN98] for rewriting, in [CDG+02] for tree automata and in [GT95] for closure of tree automata by rewriting.

Let $\mathcal{F}$ be a finite set of symbols, each associated with an arity function, and let $\mathcal{X}$ be a countable set of variables. $\mathcal{T}(\mathcal{F},\mathcal{X})$ denotes the set of terms, and $\mathcal{T}(\mathcal{F})$ denotes the set of ground terms (terms without variables). The set of variables of a term t is denoted by $\mathcal{V}ar(t)$. A substitution is a function σ from $\mathcal{X}$ into $\mathcal{T}(\mathcal{F},\mathcal{X})$, which can be extended uniquely to an endomorphism of $\mathcal{T}(\mathcal{F},\mathcal{X})$. A position p for a term t is a word over $\mathbb{N}$. The empty sequence ϵ denotes the top-most position. The set $\mathcal{P}os(t)$ of positions of a term t is inductively defined by:

– $\mathcal{P}os(t) = \{\epsilon\}$ if $t \in \mathcal{X}$
– $\mathcal{P}os(f(t_1,\ldots,t_n)) = \{\epsilon\} \cup \{i.p \mid 1 \leq i \leq n \text{ and } p \in \mathcal{P}os(t_i)\}$

If $p \in \mathcal{P}os(t)$, then $t|_p$ denotes the subterm of t at position p and $t[s]_p$ denotes the term obtained by replacement of the subterm $t|_p$ at position p by the term s. A term rewriting system (TRS) $\mathcal{R}$ is a set of *rewrite rules* $l \to r$, where $l, r \in \mathcal{T}(\mathcal{F},\mathcal{X})$, $l \notin \mathcal{X}$, and $\mathcal{V}ar(l) \supseteq \mathcal{V}ar(r)$. A rewrite rule $l \to r$ is *left-linear* if each variable of l (resp. r) occurs only once in l. A TRS $\mathcal{R}$ is left-linear if every rewrite rule $l \to r$ of $\mathcal{R}$ is left-linear). The TRS $\mathcal{R}$ induces a rewriting relation $\to_{\mathcal{R}}$ on terms whose reflexive transitive closure is denoted by $\to_{\mathcal{R}}^{\star}$. The set of $\mathcal{R}$-descendants of a set of ground terms E is $\mathcal{R}^*(E) = \{t \in \mathcal{T}(\mathcal{F}) \mid \exists s \in E \text{ s.t. } s \to_{\mathcal{R}}^{\star} t\}$.

The *verification technique* defined in [Gen98, FGVTT04] is based on $\mathcal{R}^*(E)$. Note that $\mathcal{R}^*(E)$ is possibly infinite: $\mathcal{R}$ may not terminate and/or E may be infinite. The set $\mathcal{R}^*(E)$ is generally not computable [GT95]. However, it is possible to over-approximate it [Gen98, FGVTT04, Tak04] using tree automata, i.e. a finite representation of infinite (regular) sets of terms. In this verification setting, the TRS $\mathcal{R}$ represents the system to verify, sets of terms E and Bad represent respectively the set of initial configurations and the set of "bad" configurations that should not be reached. Then, using tree automata completion, we construct a tree automaton $\mathcal{B}$ whose language $\mathcal{L}(\mathcal{B})$ is such that $\mathcal{L}(\mathcal{B}) \supseteq \mathcal{R}^*(E)$. Then if $\mathcal{L}(\mathcal{B}) \cap Bad = \emptyset$ then this proves that $\mathcal{R}^*(E) \cap Bad = \emptyset$, and thus that none of the "bad" configurations is reachable. We now define tree automata.

Let $\mathcal{Q}$ be a finite set of symbols, with arity 0, called *states* such that $\mathcal{Q} \cap \mathcal{F} = \emptyset$. $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ is called the set of *configurations*.

Definition 1 (Transition and normalized transition). *A* transition *is a rewrite rule $c \to q$, where c is a configuration i.e. $c \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ and $q \in \mathcal{Q}$. A* normalized transition *is a transition $c \to q$ where $c = f(q_1,\ldots,q_n)$, $f \in \mathcal{F}$ whose arity is n, and $q_1,\ldots,q_n \in \mathcal{Q}$.*

Definition 2 (Bottom-up nondeterministic finite tree automaton). *A bottom-up nondeterministic finite tree automaton (tree automaton for short) is a quadruple $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_F, \Delta \rangle$, where $\mathcal{Q}_F \subseteq \mathcal{Q}$ and Δ is a set of normalized transitions.*

The *rewriting relation* on $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ induced by the transitions of $\mathcal{A}$ (the set Δ) is denoted by $\rightarrow_\Delta$. When Δ is clear from the context, $\rightarrow_\Delta$ will also be denoted by $\rightarrow_\mathcal{A}$. Here is the definition of the recognized language, see [BGJ08] for examples.

Definition 3 (Recognized language). *The tree language recognized by $\mathcal{A}$ in a state q is $\mathcal{L}(\mathcal{A}, q) = \{t \in \mathcal{T}(\mathcal{F}) \mid t \rightarrow^\star_\mathcal{A} q\}$. The language recognized by $\mathcal{A}$ is $\mathcal{L}(\mathcal{A}) = \bigcup_{q \in \mathcal{Q}_F} \mathcal{L}(\mathcal{A}, q)$. A tree language is regular if and only if it can be recognized by a tree automaton.*

3 Tree Automata Completion

Given a tree automaton $\mathcal{A}$ and a TRS $\mathcal{R}$, the tree automata completion algorithm, proposed in [Gen98, FGVTT04], computes a tree automaton $\mathcal{A}^*_\mathcal{R}$ such that $\mathcal{L}(\mathcal{A}^*_\mathcal{R}) = \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ when it is possible (for some of the classes of TRSs where an exact computation is possible, see [FGVTT04]) and such that $\mathcal{L}(\mathcal{A}^*_\mathcal{R}) \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ otherwise.

The tree automata completion works as follows. From $\mathcal{A} = \mathcal{A}^0_\mathcal{R}$ completion builds a sequence $\mathcal{A}^0_\mathcal{R}.\mathcal{A}^1_\mathcal{R} \ldots \mathcal{A}^k_\mathcal{R}$ of automata such that if $s \in \mathcal{L}(\mathcal{A}^i_\mathcal{R})$ and $s \rightarrow_\mathcal{R} t$ then $t \in \mathcal{L}(\mathcal{A}^{i+1}_\mathcal{R})$. If we find a fixpoint automaton $\mathcal{A}^k_\mathcal{R}$ such that $\mathcal{R}^*(\mathcal{L}(\mathcal{A}^k_\mathcal{R})) = \mathcal{L}(\mathcal{A}^k_\mathcal{R})$, then we note $\mathcal{A}^*_\mathcal{R} = \mathcal{A}^k_\mathcal{R}$ and we have $\mathcal{L}(\mathcal{A}^*_\mathcal{R}) = \mathcal{R}^*(\mathcal{L}(\mathcal{A}^0_\mathcal{R}))$, or $\mathcal{L}(\mathcal{A}^*_\mathcal{R}) \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ if $\mathcal{R}$ is not in one class of [FGVTT04]. To build $\mathcal{A}^{i+1}_\mathcal{R}$ from $\mathcal{A}^i_\mathcal{R}$, we achieve a *completion step* which consists of finding *critical pairs* between $\rightarrow_\mathcal{R}$ and $\rightarrow_{\mathcal{A}^i_\mathcal{R}}$. To define the notion of critical pair, we extend the definition of substitutions to terms of $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$. For a substitution $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ and a rule $l \rightarrow r \in \mathcal{R}$, a critical pair is an instance $l\sigma$ of l such that there exists $q \in \mathcal{Q}$ satisfying $l\sigma \rightarrow^*_{\mathcal{A}^i_\mathcal{R}} q$ and $l\sigma \rightarrow_\mathcal{R} r\sigma$. Note that since $\mathcal{R}$, $\mathcal{A}^i_\mathcal{R}$ and the set $\mathcal{Q}$ of states of $\mathcal{A}^i_\mathcal{R}$ are finite, there is only a finite number of critical pairs. For every critical pair detected between $\mathcal{R}$ and $\mathcal{A}^i_\mathcal{R}$ such that $r\sigma \not\rightarrow^*_{\mathcal{A}^i_\mathcal{R}} q$, the tree automaton $\mathcal{A}^{i+1}_\mathcal{R}$ is constructed by adding a new transition $r\sigma \rightarrow q$ to $\mathcal{A}^i_\mathcal{R}$ such that $\mathcal{A}^{i+1}_\mathcal{R}$ recognizes $r\sigma$ in q, i.e. $r\sigma \rightarrow^*_{\mathcal{A}^{i+1}_\mathcal{R}} q$, see Figure 1.

$$\begin{tikzcd}
l\sigma \arrow[r, "\mathcal{R}"'] \arrow[d, "\mathcal{A}^i_\mathcal{R}"', "*"] & r\sigma \arrow[dl, bend left, "*"', "\mathcal{A}^{i+1}_\mathcal{R}"] \\
q &
\end{tikzcd}$$

Fig. 1. Critical pair

$$\begin{tikzcd}
u\sigma \arrow[r, equal, "\mathcal{E}"'] \arrow[d, "\mathcal{A}^{i+1}_\mathcal{R}"', "*"] & v\sigma \arrow[d, "*"', "\mathcal{A}^{i+1}_\mathcal{R}"] \\
q & q'
\end{tikzcd}$$

Fig. 2. Detection of merging

However, the transition $r\sigma \rightarrow q$ is not necessarily a normalized transition of the form $f(q_1, \ldots, q_n) \rightarrow q$ and so it has to be normalized first. Thus, instead of adding $r\sigma \rightarrow q$ we add $Norm(r\sigma \rightarrow q)$ to transitions of $\mathcal{A}^i_\mathcal{R}$. Here is the $Norm$ function used to normalize transitions. Note that, in this function, transitions are normalized using either new states of $\mathcal{Q}_{new}$ or states of $\mathcal{Q}$, states of the automaton being completed. As we will see in Lemma 1, this has no effect on the safety of the normalization but only on its precision.

Definition 4 (*$Norm$*). *Let $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ be a tree automaton, $\mathcal{Q}_{new}$ a set of* new *states such that $\mathcal{Q} \cap \mathcal{Q}_{new} = \emptyset$, $t \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ and $q \in \mathcal{Q}$. The function $Norm$ is inductively defined by:*

- *$Norm(t \to q) = \emptyset$ if $t = q$,*
- *$Norm(t \to q) = \{c \to q \mid c \to t \in \Delta\}$ if $t \in \mathcal{Q}$,*
- *$Norm(f(t_1, \ldots, t_n) \to q) = \bigcup_{i=1\ldots n} Norm(t_i \to q_i) \cup \{f(q_1, \ldots, q_n) \to q\}$ where $\forall i = 1 \ldots n : (t_i \in \mathcal{Q} \Rightarrow q_i = t_i) \wedge (t_i \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q}) \setminus \mathcal{Q} \Rightarrow q_i \in \mathcal{Q} \cup \mathcal{Q}_{new})$.*

When using only new states to normalize all the new transitions occurring in all the completion steps, completion is as precise as possible. However, doing so, completion is likely not to terminate (because of general undecidability results [GT95]). Enforcing termination of completion can be easily done by bounding the set of new states to be used with $Norm$ during the whole completion. We then obtain a finite tree automaton over-approximating the set of reachable states. The fact that normalizing with any set of states (new or not) is *safe* is guaranteed by the following simple lemma. For the general safety theorem of completion see [FGVTT04].

Lemma 1. *For all tree automaton $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$, $t \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q}) \setminus \mathcal{Q}$ and $q \in \mathcal{Q}$, if $\Pi = Norm(t \to q)$ whatever the states chosen in $Norm(t \to q)$ we have $t \to_{\Pi}^{*} q$.*

Proof. This can be done by a simple induction on transitions [FGVTT04].

To let the user of completion guide the approximation, we use two different tools: a set N of *normalization rules* (see [FGVTT04]) and a set $\mathcal{E}$ of *approximation equations*. Rules and equations can be either defined by hand so as to prove a complex property [GTTVTT03], or generated automatically when the property is more standard [BHK04]. Normalization rules can be seen as a specific strategy for normalizing new transitions using the $Norm$ function. We have seen that Lemma 1 is enough to guarantee that the chosen normalization strategy has no impact on the safety of completion. Similarly, for our checker, we will see in Section 8 that the related Coq safety proof can be carried out independently of the normalization strategy (i.e. set N of normalization rules). On the opposite, the effect of approximation equations is more complex and has to be studied more carefully. An approximation equation is of the form $u = v$ where $u, v \in \mathcal{T}(\mathcal{F}, \mathcal{X})$. Let $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ be a substitution such that $u\sigma \to_{\mathcal{A}_{\mathcal{R}}^{i+1}} q$, $v\sigma \to_{\mathcal{A}_{\mathcal{R}}^{i+1}} q'$ and $q \neq q'$, see Figure 2. Then, we know that there exists some terms recognized by q and some recognized by q' which are equivalent modulo $\mathcal{E}$. A correct over-approximation of $\mathcal{A}_{\mathcal{R}}^{i+1}$ consists in applying the $Merge$ function to it, i.e. replace $\mathcal{A}_{\mathcal{R}}^{i+1}$ by $Merge(\mathcal{A}_{\mathcal{R}}^{i+1}, q, q')$, as long as an approximation equation of $\mathcal{E}$ applies. The $Merge$ function, defined below, merges states in a tree automaton. See [BGJ08] for examples of completion and approximation.

Definition 5 (*$Merge$*). *Let $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_F, \Delta \rangle$ be a tree automaton and q_1, q_2 be two states of $\mathcal{A}$. We denote by $Merge(\mathcal{A}, q_1, q_2)$ the tree automaton where every occurrence of q_2 is replaced by q_1 in $\mathcal{Q}$, $\mathcal{Q}_F$ and in every left-hand side and right-hand side of every transition of Δ.*

4 A Result Checker for Tree Automata Completion

By moving the certification problem from the completion algorithm to the checker, the certification problem consists in proving the following Coq theorem. In Coq specifications, recall that '→' is used to denote both the logical implication and functional types. Similarly, ':' is used to give the type of a function, the type of a data or the statement for a theorem.

```
Theorem sound_checker :
      ∀ A A' R, checker A R A' = true → ApproxReachable A R A'.
```

where `ApproxReachable` is a Coq predicate that describes the Soundness Property: $\mathcal{L}(A')$ *contains all terms reachable by rewriting terms of* $\mathcal{L}(A)$ *with* $\mathcal{R}$, i.e. $\mathcal{L}(\mathcal{A}') \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$. To state formally this predicate in Coq, we need to give a Coq axiomatization of Term Rewriting Systems and of Tree Automata. It is given in Section 5. Given two automata $\mathcal{A}$, $\mathcal{A}'$ and a TRS $\mathcal{R}$ the checker verifies that $\mathcal{L}(\mathcal{A}') \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ or (`ApproxReachable A R A'`) in Coq. To perform this, we need to check the two following properties:

- `Included`: inclusion of initial set in the fixpoint: $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A}')$.
- `IsClosed`: $\mathcal{A}'$ is closed by rewriting with $\mathcal{R}$: For all $l \to r \in \mathcal{R}$ and all $t \in \mathcal{L}(\mathcal{A}')$, if t is rewritten in t' by the rule $l \to r$ then $t' \in \mathcal{L}(\mathcal{A}')$.

For each item, we provide a Coq function and its correctness theorem: function `inclusion` is dedicated to inclusion checking and function `closure` checks if a tree automaton is closed by rewriting. We also give the theorem used to deduce `ApproxReachable A R A'` from `Included A A'` and `IsClosed R A'`:

```
Theorem inclusion_sound:
      ∀ A A', inclusion A A' = true → Included A A'.

Theorem closure_sound:
      ∀ R A', closure R A' = true → IsClosed R A'.

Theorem Included_IsClosed_ApproxReachable:
      ∀ A A' R, Included A A' → IsClosed R A' →
          ApproxReachable A R A'.
```

Note that, in this paper we focus on the proof of $\mathcal{L}(\mathcal{A}') \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$. However, to prove the unreachability property, the emptiness of the intersection between $\mathcal{L}(\mathcal{A}')$ and the bad term set has also to be verified. Since the formalization in Coq of the intersection and emptiness decision are close to their standard definition [CDG+02], and since they have already been covered by [RGL01], they are not be detailed in this paper.

5 Formalization of Term Rewriting Systems

The aim of this part is to formalize in Coq: terms, term rewriting systems, reachable terms and the reachability problem itself. First, we use the positive integers provided by the Coq's standard library to define symbol sets like variables ($\mathcal{X}$) or function symbols ($\mathcal{F}$). We rename `positive` into `ident` to be more explicit. Then, we define term set $\mathcal{T}(\mathcal{F}, \mathcal{X})$ using inductive types:

```
Inductive term : Set :=
| Fun : ident → list term → term
| Var : ident → term.
```

A rewrite rule $l \to r$ is represented by a pair of terms with a well-definition proof, i.e. a Coq proof that the set of variables of r is a subset of the set of variables of l. The function `Fv : term →list ident` builds the set of variables for a term.

```
Inductive rule : Set :=
| Rule (l r : term)(H : subseteq (Fv r) (Fv l)) : rule.
```

In the following, `list rule` type represents a TRS. In Coq we use `(t @ sigma)` to denote the term resulting of the application of a substitution `sigma` to each variable that occurs in a term `t`. In Coq, the rewriting relation *"u is rewritten in v by $l \to r$"*, commonly defined by $\exists \sigma\ s.t.\ u|_p = l\sigma \ \wedge\ v = u[r\sigma]_p$, is split into two predicates:

– The first one defines the rewriting of a term at the topmost position. In fact, the set of term pairs (t, t') which are rewritten at the top most by the rule can be seen as the set of term pairs $(l\sigma, r\sigma)$ for any substitution σ.
– The second one just defines inductively the rewriting relation at any position of a term t by a rule $l \to r$, by the topmost rewriting of any subterm of t by $l \to r$.

```
(* Topmost rewriting : *)
Inductive TRew (x : rule) : term → term → Prop :=
| R_Rew : ∀ s l r (H : subseteq (Fv r) (Fv l)),
           x = Rule l r H → TRew x (l @ s) (r @ s).
```

Similarly, using an inductive definition it is possible to define the `Rew` predicate for rewriting at any position. Then we have to define $\to_{\mathcal{R}}^*$. In Coq, we prefer to see it as the predicate `Reachable R u` that characterizes the set of reachable terms from u by $\to_{\mathcal{R}}^*$.

```
Inductive Reachable(R : list rule)(t : term) : term → Prop:=
| R_refl : Reachable R t t
| R_trans : ∀ u v r, Reachable R t u → In r R → Rew r u v →
    Reachable R t v.
```

6 Formalization of Tree Automata

The fact that the checker, to be executed, is directly extracted from the Coq formalization has an important consequence on the tree automata formalization. Since the data structures used in the formalization are those that are really used for the execution, they need to be formal *and* efficient. For tree automata, instead of a naive representation, it is thus necessary to use optimized formal data structures borrowed from [RGL01]. In Section 5, we have represented variables $\mathcal{X}$ and function symbols $\mathcal{F}$ by the type `ident`. We do the same for $\mathcal{Q}$. We define a tree automaton as a pair $(\mathcal{Q}_F, \Delta)$, where $\mathcal{Q}_F$ is the finite set of final states, and Δ the finite set of normalized transitions like $f(q_1, \ldots, q_n) \to q$. In Coq, the `t_automaton` record stands for this pair where the final state field `qf` is a simple `list ident` and the transition set field `delta` has `Delta.t` type. The `Delta` module contains the implementation, the theorems and proofs for normalized transition sets. This representation is based on the `FMapPositive` Coq library of functional mappings, where data are indexed by `positive` numbers. A set of transitions of type `Delta.t` is a map where each state q indexes a set of configuration like $f(q_1, \ldots, q_n)$. In the same way, each set of configuration is a map where function symbols are used to index a stack of state lists. For all transitions $f(q_1, \ldots, q_n) \to q$, the state list $(q_1, \ldots, q_n)$ is stored in the stack indexed by symbol f and state q. Now we can define a predicate to characterize the recognized language of a tree automaton. In fact, we are defining the set of ground terms that are reduced to a state q by a transition set Δ. This set, which corresponds to $\mathcal{L}(\mathcal{A}, q)$ if Δ is the set of transitions of $\mathcal{A}$, can be constructed inductively in Coq using the single deduction rule:

$$\frac{t_1 \in \mathcal{L}(\Delta, q_1) \qquad \ldots\ldots \qquad t_n \in \mathcal{L}(\Delta, q_n)}{f(t_1, \ldots, t_n) \in \mathcal{L}(\Delta, q)} \text{ If } f(q_1, \ldots, q_n) \to q \in \Delta$$

In Coq, we express this statement using a mutually inductive predicate `IsRec`. A term t is recognized by a tree automaton $(\mathcal{Q}_F, \Delta)$, if the predicate `IsRec` $\Delta\ q\ t$ is valid for $q \in \mathcal{Q}_F$.

7 An Optimized Inclusion Checker

In this part, we give the formal definition of the `Included` property and of the `inclusion` Coq function used to effectively check the tree automata inclusion. From the previous formal definitions on tree automata, we can state the `Included` predicate in the following way:

```
Definition Included (a b : t_automaton) : Prop :=
  ∀ t q, In q a.(qf) → IsRec a.(delta) q t →
    ∃ q', In q' b.(qf) ∧ IsRec b.(delta) q' t.
```

Now let us focus on the function `inclusion` itself. The usual algorithm for proving inclusion of regular languages recognized by nondeterministic bottom-up tree automata, for instance for proving $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$, consists in proving that $\mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\overline{\mathcal{B}}) = \emptyset$, where $\overline{\mathcal{B}}$ is the complement automaton for $\mathcal{B}$. However, the algorithm for building $\overline{\mathcal{B}}$ from $\mathcal{B}$ is EXPTIME-complete [CDG+02]. This is the reason why we here define a criterion with a better practical complexity. It is is based on a simple syntactic comparison of transition sets, i.e. we check the inclusion of transition sets modulo the renamings performed by the *Merge* function. This increases a lot the efficiency of our checker, especially by saving memory. This is crucial to check inclusion of big tree automata (see Section 9).

This algorithm is correct but, of course, it is not complete in general, i.e. not always able to prove that $\mathcal{L}(\mathcal{A}) \not\subseteq \mathcal{L}(\mathcal{B})$. However, we show in the following that, under certain conditions on $\mathcal{A}$ and $\mathcal{B}$ which are satisfied if $\mathcal{B}$ is obtained by completion of $\mathcal{A}$, this algorithm is also complete and thus becomes a decision procedure. First, we introduce the following notation:

Γ	: induction hypothesis set
Δ_i	: transition set of the tree automaton $\mathcal{A}_i$
$\{c \mid c \to q \in \Delta\}$	: configurations of Δ that are rewritten in q
$\{c_i\}_n^m$	: configuration set from c_n to c_m

We formulate our inclusion problem by formulas of the form: $\Gamma \vdash_{A,B} q \Subset q'$. Such a statement stands for: under the assumption Γ, it is possible to prove that $\mathcal{L}(A, q) \subseteq \mathcal{L}(B, q')$. The algorithm consists in building proof trees for those statements using the following set of deduction rules.

$$\text{(Induction)}\ \frac{\Gamma \cup \{q \Subset q'\} \vdash_{A,B} \{c \mid c \to_{\Delta_A} q\} \Subset \{c \mid c \to_{\Delta_B} q'\}}{\Gamma \vdash_{A,B} q \Subset q'}\ \text{if } (q \Subset q') \notin \Gamma$$

$$\text{(Axiom)}\ \frac{}{\Gamma \cup \{q \Subset q'\} \vdash_{A,B} q \Subset q'} \qquad \text{(Empty)}\ \frac{}{\Gamma \vdash_{A,B} \emptyset \Subset \{c'_j\}_1^m}$$

$$\text{(Split-l)}\ \frac{\Gamma \vdash_{A,B} c_1 \Subset \{c'_j\}_1^m \quad \ldots\ldots \quad \Gamma \vdash_{A,B} c_n \Subset \{c'_j\}_1^m}{\Gamma \vdash_{A,B} \{c_i\}_1^n \Subset \{c'_j\}_1^m}$$

$$\text{(Weak-r)}\ \frac{\Gamma \vdash_{A,B} c \Subset c'_k}{\Gamma \vdash_{A,B} c \Subset \{c'_i\}_1^n}\ \text{if } (1 \le k \le n) \qquad \text{(Const.)}\ \frac{}{\Gamma \vdash_{A,B} a() \Subset a()}$$

$$\text{(Config)}\ \frac{\Gamma \vdash_{A,B} q_1 \Subset q'_1 \quad \ldots\ldots \quad \Gamma \vdash_{A,B} q_n \Subset q'_n}{\Gamma \vdash_{A,B} f(q_1, \ldots, q_n) \Subset f(q'_1, \ldots, q'_n)}$$

Given $\mathcal{Q}_{F_\mathcal{A}}$ and $\mathcal{Q}_{F_\mathcal{B}}$ the sets of final states of $\mathcal{A}$ and $\mathcal{B}$, $\#()$ a symbol of arity 1 not occurring in $\mathcal{F}$, to prove $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$, we start our deduction by the statement: $\emptyset \vdash_{\mathcal{A},\mathcal{B}} \{\#(q) \mid q \in \mathcal{Q}_{F_\mathcal{A}}\} \Subset \{\#(q) \mid q \in \mathcal{Q}_{F_\mathcal{B}}\}$

Example 1. Let $\mathcal{A}$ and $\mathcal{B}$ be two automata s.t.:

$$\mathcal{A} = \left\{ \begin{array}{r} a \to q_1 \\ b \to q_2 \\ f(q_1, q_2) \to \mathbf{q} \end{array} \right\} \text{ with } \mathcal{Q}_{F_\mathcal{A}} = \{\mathbf{q}\} \text{ and } \mathcal{B} = \left\{ \begin{array}{r} a \to \mathbf{q}' \\ b \to \mathbf{q}' \\ f(q', q') \to \mathbf{q}' \end{array} \right\} \text{ with } \mathcal{Q}_{F_\mathcal{B}} = \{\mathbf{q}'\}$$

Here we have $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$ and we can derive $\emptyset \vdash_{\mathcal{A},\mathcal{B}} \#(q) \Subset \#(q')$ with the deduction rules:

$$\text{(Config)}\ \dfrac{\text{(Induction)}\ \dfrac{\text{(Weark-r)}\ \dfrac{\text{(Config)}\ \dfrac{\text{(Ind.)}\ \dfrac{\text{(W-r)}\ \dfrac{\text{(Const.)}\ \dfrac{}{\{q \Subset q', q_1 \Subset q'\} \vdash_{\mathcal{A},\mathcal{B}} a() \Subset a()}}{\{q \Subset q',\ q_1 \Subset q'\} \vdash_{\mathcal{A},\mathcal{B}} a() \Subset \{a(), b(), f(q', q')\}}}{\{q \Subset q'\} \vdash_{\mathcal{A},\mathcal{B}} q_1 \Subset q'} \quad \dfrac{\dfrac{\text{(Const.)}\ \dfrac{}{\{q \Subset q', q_2 \Subset q'\} \vdash_{\mathcal{A},\mathcal{B}} b() \Subset b()}}{\{q \Subset q',\ q_2 \Subset q'\} \vdash_{\mathcal{A},\mathcal{B}} b() \Subset \{a(), b(), f(q', q')\}}}{\{q \Subset q'\} \vdash_{\mathcal{A},\mathcal{B}} q_2 \Subset q'}}{\{q \Subset q'\} \vdash_{\mathcal{A},\mathcal{B}} f(q_1, q_2) \Subset f(q', q')}}{\{q \Subset q'\} \vdash_{\mathcal{A},\mathcal{B}} f(q_1, q_2) \Subset \{a(), b(), f(q', q')\}}}{\emptyset \vdash_{\mathcal{A},\mathcal{B}} q \Subset q'}}{\emptyset \vdash_{\mathcal{A},\mathcal{B}} \#(q) \Subset \#(q')}$$

The main property we want to demonstrate in Coq is that this syntactic criterion implies the semantic inclusion for the considered languages in 6.

```
Theorem inclusion_sound :
   ∀ A B, inclusion A B = true → Included A B.
```

Before proving this in Coq, we need to define more formally the function `inclusion`. This function cannot be defined as a simple structural recursion. Thus Coq needs a termination proof for this algorithm. Thanks to the Coq feature `Function`, it is possible to define the algorithm using a measure function and provide a proof that its value decreases at each recursive call to ensure the termination.

Termination of deduction rules can be proved by defining a measure function μ on statements of the form $\Gamma \vdash_{\mathcal{A},\mathcal{B}} \alpha \Subset \beta$. The Γ relation can be seen as a subset of $\mathcal{Q}_A \times \mathcal{Q}_B$ which is a finite set. All tree automata have a finite number of states. Then the statement measure $\mu(\Gamma \vdash_{A,B} \alpha \trianglelefteq \beta)$ is defined as tuple $(\mu_1(\Gamma), \mu_2(\alpha) + \mu_2(\beta))$ where:

$$\left[\begin{array}{l} \mu_1(\Gamma) = |\mathcal{Q}_A \times \mathcal{Q}_B| - |\Gamma| \\ \mu_2(x) \;= \begin{cases} (m+1-n) \text{ if } x = \{c_i\}_n^m \\ 1 \text{ if } x = f(q_1, \ldots, q_n), \\ 0 \text{ otherwise} \end{cases} \end{array}\right.$$

Then we define the ordering $\ll$ by the lexicographic combination of the usual order $<$ on natural numbers for μ_1 and μ_2. Since $<$ is well founded, the lexicographic combination $\ll$ is also well founded.

Theorem 1. *(Termination) At each step, the measure decreases strictly:*

$$\frac{\Gamma_1 \vdash_{A,B} \alpha_1 \Subset \beta_1 \ldots \Gamma_n \vdash_{A,B} \alpha_n \Subset \beta_n}{\Gamma \vdash_{A,B} \alpha \Subset \beta} \Longrightarrow \bigwedge_{i=1}^{n} \mu(\Gamma_i \vdash_{A,B} \alpha_i \Subset \beta_i) \ll \mu(\Gamma \vdash_{A,B} \alpha \Subset \beta)$$

Proof. See [BGJ08], for details.

Theorem 2. *(Soundness) For all tree automata $\mathcal{A}$ and $\mathcal{B}$, if there exists $\prod$ a proof tree of $\emptyset \vdash_{\mathcal{A},\mathcal{B}} q \Subset q'$ then we have $\mathcal{L}(\Delta_{\mathcal{A}}, q) \subseteq \mathcal{L}(\Delta_{\mathcal{B}}, q')$*

Proof. This can be done by an induction on the size of the term of $\mathcal{L}(\Delta_{\mathcal{A}}, q)$. See [BGJ08] for details.

As said above, the described algorithm is not complete in general. However, we show that it is complete for tree automata produced by completion. In particular if $\mathcal{A}^k_{\mathcal{R}}$ is obtained after k completion step from $\mathcal{A}^0$ then we can build a proof $\prod$ for the statement $\emptyset \vdash_{\mathcal{A}^0,\mathcal{A}^k_{\mathcal{R}}} \{\#(q) \mid q \in \mathcal{Q}_{F_0}\} \Subset \{\#(q') \mid q' \in \mathcal{Q}_{F_k}\}$. Recall that the tree automaton produced by the k^{th} step of completion is noted $\mathcal{A}_k = \langle \mathcal{F}, \mathcal{Q}_k, \mathcal{Q}_{F_k}, \Delta_k \rangle$. The tree automata completion performs two main operations at each step of calculus: *normalization* and *state merging*. In the case of normalization, the language inclusion can simply be proved using transition set inclusion. With the state merging operation, set inclusion is not enough because it implies transition merging too. This is the reason why we have to define a new order relation preserved by each operation.

Definition 6. *Given $\mathcal{A}$, $\mathcal{B}$ two tree automata, $\sqsubseteq$ is the reflexive and transitive relation defined as follows: $\mathcal{A} \sqsubseteq \mathcal{B}$ if there exists a function ϱ that renames states of $\mathcal{A}$ into states of $\mathcal{B}$ and such that all renamed rules $\Delta_{\mathcal{A}}$ are contained in $\Delta_{\mathcal{B}}$:*

$$\mathcal{A} \sqsubseteq \mathcal{B} \iff \exists \varrho : \mathcal{Q}_{\mathcal{A}} \to \mathcal{Q}_{\mathcal{B}},\ \varrho(\Delta_{\mathcal{A}}) \subseteq \Delta_{\mathcal{B}}\ \wedge\ \varrho(\mathcal{Q}_{F_{\mathcal{A}}}) \subseteq \mathcal{Q}_{F_{\mathcal{B}}} \tag{1}$$

Lemma 2. *Given a tree automaton $\mathcal{A}$,*

1. *if $\mathcal{A}' = \mathcal{A} \cup Norm(r\sigma \to q)$ then $\mathcal{A} \sqsubseteq \mathcal{A}'$*
2. *if $\mathcal{A}' = Merge(\mathcal{A}, q_1, q_2)$ then $\mathcal{A} \sqsubseteq \mathcal{A}'$*

Proof. For details, see [BGJ08].

Theorem 3. *Given a tree automaton $\mathcal{A}^0$, a TRS $\mathcal{R}$ and an equation set $\mathcal{E}$, after k completion steps we obtain $\mathcal{A}^k_{\mathcal{R}}$ such that $\mathcal{A}^0 \sqsubseteq \mathcal{A}^k_{\mathcal{R}}$.*

Proof. Since we have proved that $\sqsubseteq$ is preserved by $Norm$ and $Merge$ functions, it is also the case for every completion step between $\mathcal{A}^k_{\mathcal{R}}$ and $\mathcal{A}^{k+1}_{\mathcal{R}}$, i.e $\mathcal{A}^k_{\mathcal{R}} \sqsubseteq \mathcal{A}^{k+1}_{\mathcal{R}}$. Then, the theorem can be deduced using the reflexivity and transitivity of $\sqsubseteq$. See [BGJ08].

Now, we define the completeness property as the following:

Theorem 4. *(Completeness) Given two tree automata $\mathcal{A}$ and $\mathcal{B}$ if $\mathcal{A} \sqsubseteq \mathcal{B}$ then there exists $\prod$ a proof of statement $\emptyset \vdash_{\mathcal{A},\mathcal{B}} \{\#(q_f) \mid q_f \in \mathcal{Q}_{F_A}\} \Subset \{\#(q'_f) \mid q'_f \in \mathcal{Q}_{F_B}\}$.*

Proof. For details, see [BGJ08].

Thus, we can ensure that for an automaton $\mathcal{A}^k_{\mathcal{R}}$ obtained by k completion steps from $\mathcal{A}^0$, there exists a proof $\prod$ of the statement $\emptyset \vdash_{\mathcal{A}^0,\mathcal{A}^k_{\mathcal{R}}} \{\#(q) \mid q \in \mathcal{Q}_{F_0} \Subset \{\#(q') \mid q' \in \mathcal{Q}_{F_k}\}$. This can be obtained by a simple combination of the two previous theorems.

Finally, as shown in [BGJ08], this algorithm has a polynomial complexity w.r.t. space. Using tabling, it can also be implemented so as to be polynomial in time. However, since proofs are more difficult to carry out in Coq on a tabled version, we chose to stick to a simpler implementation that appears to be sufficient for our test cases (see Section 9).

8 Formalization of Closure by Rewriting

In this part we aim at defining formally the `IsClosed` predicate, the function `closure` and prove the soundness of this function w.r.t. `IsClosed`. Recall that to check if a tree automaton $\mathcal{A} = \langle \mathcal{Q}_F,\ \Delta \rangle$ is closed w.r.t. a TRS $\mathcal{R}$, it is enough to prove that for all $t \in \mathcal{L}(\mathcal{A})$, if t' is reachable from t by $\to^*_{\mathcal{R}}$ then $t' \in \mathcal{L}(\mathcal{A})$. Now that we have defined in Coq rewriting and tree automata, we can define more formally the `IsClosed` predicate and recall the `closure_sound` theorem to prove:

```
Definition IsClosed (R : list rule) (A : t_aut) : Prop :=
   ∀ q t t', IsRec A.(delta) q t → Reachable R t t' →
       IsRec A.(delta) q t'.

Theorem closure_sound: ∀ R A', LeftLinear R →
   closure R A' = true → IsClosed R A'.
```

The algorithm to check closure of $\mathcal{A}$ by $\mathcal{R}$ computes for each left-linear rule $l \to r \in \mathcal{R}$ the full set of the substitutions σ s.t. $l\sigma \to_{\Delta}^{*} q$ and then, checks that $r\sigma \to_{\Delta}^{*} q$. Then, the correctness proof consists in showing that if `closure` answers true, then $\mathcal{L}(\mathcal{A})$ is closed by $\to_{\mathcal{R}}$. We now give some hints to define the `closure` function. First, for all left-linear rule $l \to r$ of $\mathcal{R}$, this function has to find all the substitutions $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ and all the states $q \in \mathcal{Q}$ such that $l\sigma \to_{\Delta}^{*} q$. This is what we call the *matching-problem*. Second, this function has to check that for all the q and σ found, we have $r\sigma \to_{\Delta}^{*} q$. Third, in the correctness theorem, we have to show that all the substitutions $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ cover the set of substitutions on terms, i.e. of the form $\sigma' : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F})$, and hence cover all reachable terms.

We note $l \trianglelefteq q$ the matching problem consisting in finding all the substitutions $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ and all the states $q \in \mathcal{Q}$ such that $l\sigma \to_{\Delta}^{*} q$. An algorithm solving this kind of problems was defined in [Gen97]. Note that it is complete only if l is linear. The algorithm consists in normalizing the formula $l \trianglelefteq q$ with the following deduction rules:

$$\text{(Unfold)}\ \frac{f(s_1,\ldots,s_n) \trianglelefteq f(q_1,\ldots,q_n)}{s_1 \trianglelefteq q_1 \wedge \cdots \wedge s_n \trianglelefteq q_n} \qquad \text{(Clash)}\ \frac{f(s_1,\ldots,s_n) \trianglelefteq g(q'_1,\ldots q'_m)}{\perp}$$

$$\text{(Config)}\ \frac{s \trianglelefteq q}{s \trianglelefteq c_1 \vee \cdots \vee s \trianglelefteq c_k \vee \perp} \text{if } s \notin \mathcal{X}, \text{ and } \forall c_i,\ s.t.\ c_i \to q \in \Delta.$$

Moreover, after each application of one of these rules, the result is also rewritten into disjunctive normal form. When normalization of the initial problem is terminated, we obtain a formula like $\bigvee_{i=1}^{n} \phi_i$ where $\phi_i = \bigwedge_{j=1}^{m} x_j^i \trianglelefteq q_j^i$ such that $x_j^i \in \mathcal{X}$ and $q_j^i \in \mathcal{Q}$. Each ϕ_i can be seen as a substitution $\sigma_i = \{x_j^i \mapsto q_j^i\}$. The implementation of the `matching` function in Coq is very close to this algorithm. Moreover, the soundness and completeness properties of this algorithm can be defined in Coq as follows:

```
Theorem matching_sound : ∀ D q l s, In s (matching D q l) →
    IsRed D q (l @ s).

Theorem matching_complete : ∀ D q l s, linear l →
    IsRed D q (l @ s) → In s (matching D q l).
```

As mentioned before, the `matching` algorithm is only complete for linear terms. Thus, this assumption occurs in the `matching_complete` theorem as well as in all theorems using the left-side of a rule. The second part of the `closure` function consists in verifying that for each substitution σ s.t. $l\sigma \to_{\Delta}^{*} q$, we also have $r\sigma \to_{\Delta}^{*} q$. This job is performed using the `all_red` function, we define, whose purpose is to check that this property is true for all the found substitutions. Then, we only need to prove the soundness of this function using the following Coq theorem:

```
Theorem all_red_sound : ∀ D q r sigmas,
   all_red D q r sigmas = true →
      ∀ s, In s sigmas → IsRed D q (r@s).
```

By combining the `matching` and the `all_red` functions, we obtain the function `closure_at_state` for checking up all critical pairs found at state q and for the rule $l \to r$. We define the combination as:

```
Definition closure_at_state D q l r :=
   all_red D q r (matching D q l).
Theorem closure_at_state_sound : ∀ D q l r,
   linear l → closure_at_state D q l r = true →
      (∀ s, IsRed D q (l @ s) → IsRed D q (r @ s)).
```

Given a left-linear rule $l \to r$ and a state q, this algorithm answers `true` if for all substitution $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ s.t. $l\sigma \to_{\Delta}^{*} q$ then $r\sigma \to_{\Delta}^{*} q$. Now that we have proved this result for substitutions $\sigma : \mathcal{X} \mapsto \mathcal{Q}$, we have to prove that it implies the same property for substitutions $\sigma' : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F})$, this is Lemma 3. On the opposite, to prove that every reachable term of $\mathcal{T}(\mathcal{F})$ will be covered by a configuration of $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ in Δ, we have to prove that if there exists a substitution $\sigma' : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F})$, then we can construct a corresponding substitution $\sigma : \mathcal{X} \mapsto \mathcal{Q}$, this is Lemma 4.

Lemma 3. *Given a term $u \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ a substitution s.t. $u\sigma \to_{\Delta}^{*} q$, if we have a substitution $\sigma' : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F})$ s.t. $\forall x \in \mathcal{D}om(\sigma) : \sigma' x \in \mathcal{L}(\Delta, \sigma x)$, then we have $u\sigma' \to_{\Delta}^{*} q$ and thus $u\sigma' \in \mathcal{L}(\Delta, q)$.*

Roughly, if the left or right-hand side u of a rewriting rule matches a configuration $u\sigma \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ and $u\sigma \to_{\Delta}^{*} q$ then, all terms $u\sigma' \in \mathcal{T}(\mathcal{F})$, matched by u, are also reducible into q, i.e. $u\sigma' \to_{\Delta}^{*} q$ and $u\sigma' \in \mathcal{L}(\Delta, q)$.

Lemma 4. *Given a term $u \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, if there exists a substitution $\sigma' : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F})$ such that $u\sigma' \to_{\Delta}^{*} q$, then there exists a substitution $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ s.t. $\sigma' x \in \mathcal{L}(\Delta, \sigma x)$ and $u\sigma \to_{\Delta}^{*} q$.*

Using those two lemmas, we can conclude that for all term $t \in \mathcal{L}(\Delta, q)$ rewritten in t' at the topmost position by $l \to r$, then $t' \in \mathcal{L}(\Delta, q)$. This property is easily lifted as a property of the `closure` function for all states of $\mathcal{Q}$ and using all rules of $\mathcal{R}$ at topmost position. Then, it is enough to lift this property to general rewriting at any position. Finally, the `closure_sound` general theorem is shown by using a reflexive and transitive application of the last property.

9 Benchmarks

From the Coq formal specification (about 2000 lines for definitions and 5500 lines for proofs), we have extracted an Ocaml checker implementation which is connected to the Timbuk parser. Since Coq extraction ignore all Ocaml data types (integers, lists, maps...) and redefine all them (including primitive types). Thus, we defined a set of functions to convert Coq types into Ocaml types and conversely.

In the following table, we have collected several benchmarks. For each test, we give the size of the two tree automata (initial $\mathcal{A}^0$ and completed $\mathcal{A}^*_{\mathcal{R}}$) as number of transitions/number of states. For each TRS $\mathcal{R}$ we give the number of rules. The 'CS' column gives the number of completion steps necessary to complete $\mathcal{A}^0$ into $\mathcal{A}^*_{\mathcal{R}}$ and 'CT' gives the completion time. The 'CKT' column gives the time for the checker to certify the $\mathcal{A}^*_{\mathcal{R}}$ and the 'CKM' gives the memory usage. The important thing to observe here is that, the completion time is very long (sometimes more than 24 hours), the *checking* of the corresponding automaton is always fast (a matter of seconds).

The four tests are Java programs translated into term rewriting systems using the technique detailed in [BGJL07]. All of them are completed using Timbuk except the example `List2.java` which has been completed using a new optimized completion tool detailed in [BBGM08]. In this last paper, the completion times are 10 to 100 times better than using Timbuk. Even if the input and output of this tool are tree automata, the internal computation mechanism is exclusively based on term rewriting and uses no tree automata algorithms. This shows that the completed automaton produced by a totally different algorithm and fully optimized tool is also accepted by our checker. The `List1.java` and `List2.java` corresponds to the same Java program but with slightly different encoding into TRS and approximations. The `Ex_poly.java` is the example given in [BGJL07] and the `Bad_Fixp` is the same problem as `Ex_poly.java` except that the completed automaton $\mathcal{A}^*_{\mathcal{R}}$ has been intentionally corrupted. Thus, this is thus not a valid fixpoint and rejected by the checker.

Name	$\mathcal{A}^0$	$\mathcal{A}^*_{\mathcal{R}}$	$\mathcal{R}$	CS	CT	CKT	CKM
`List1.java`	118/82	422/219	228	180	≈ 3 days	0,9s	2,3 Mo
`List2.java`	1/1	954/364	308	473	1h30	2,2s	3,1 Mo
`Ex_poly.java`	88/45	951/352	264	161	≈ 1 day	2,5s	3,3 Mo
`Bad_Fixp`	88/45	949/352	264	161	≈ 1 day	1,6s	3,2 Mo

10 Conclusion and Further Research

In this paper we have defined a Coq checker for tree automata completion. The first characteristic of the work presented here is that the checker does not validate a specific implementation of completion but, instead, the result. As a consequence, the checker remains valid even if the implementation of the completion algorithm changes or is optimized. For example, this checker could be used to certify tree automata produced by [Jac96, Ret99] and [Tak04] for left-linear TRS, provided that ϵ-transitions are normalized first. This is quite natural since the behavior of those algorithms is close to tree automata completion. We gave an even more significant example of the independence of the checker w.r.t. the used completion algorithm by certifying results produced by [BBGM08] whose algorithm is not based on tree automata.

A second salient feature of the checker is that its code is directly generated from the correctness proof of its Coq specification through the extraction mechanism. Third, we have payed particular attention to the formalization of the checker in order not to lose efficiency to obtain the certification. We have defined a specific inclusion algorithm in order to avoid the usual exponential blow-up obtained with the standard inclusion algorithm. We have defined the Coq formal specification so that it was possible to extract an independent OCaml checker. Finally, we made an extensive use of efficient formal data structures leading to more complex proof but also to faster extracted checker. An extension for non left-linear TRS, which are necessary for specifying cryptographic protocols, is under development. Since many different kinds of analyzes can be expressed as reachability problems over tree automata, and since verification of completed automata revealed to be very efficient, we aim at using this technique in a PCC architecture where tree automata are viewed as program certificates. At last, note that even if this checker is external to Coq, we can use the correction proof of the checker and the Coq reflexivity mechanism to lift-up the external verification into a proof in the Coq system. This can be necessary to perform efficient unreachability proofs on rewriting systems in Coq using an external completion tool.

References

[ABB+05] Armando, A., Basin, D., Boichut, Y., Chevalier, Y., Compagna, L., Cuellar, J., Hankes Drielsma, P., Héam, P.-C., Kouchnarenko, O., Mantovani, J., Mödersheim, S., von Oheimb, D., Rusinowitch, M., Santos Santiago, J., Turuani, M., Viganò, L., Vigneron, L.: Natural deduction with general elimination rules. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 541–567. Springer, Heidelberg (2005)

[BBGM08] Ballad, E., Boichut, Y., Genet, T., Moreau, P.-E.: Towards an Efficient Implementation of Tree Automata Completion. In: AMAST 2008 (to be published, 2008)

[BC04] Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. Springer, Heidelberg (2004)

[BD04] Barthe, G., Dufay, G.: A tool-assisted framework for certified bytecode verification. In: Wermelinger, M., Margaria-Steffen, T. (eds.) FASE 2004. LNCS, vol. 2984, pp. 99–113. Springer, Heidelberg (2004)

[BGJ08] Boyer, B., Genet, T., Jensen, T.: Certifying a Tree Automata Completion Checker. Technical Report RR 6462, INRIA (2008), http://hal.inria.fr/inria-00258275/fr/

[BGJL07] Boichut, Y., Genet, T., Jensen, T., Leroux, L.: Rewriting Approximations for Fast Prototyping of Static Analyzers. In: Baader, F. (ed.) RTA 2007. LNCS, vol. 4533, pp. 48–62. Springer, Heidelberg (2007)

[BHK04] Boichut, Y., Héam, P.-C., Kouchnarenko, O.: Automatic Approximation for the Verification of Cryptographic Protocols. In: Proc. AVIS 2004, joint to ETAPS 2004, Barcelona (Spain) (2004)

[BJP06] Besson, F., Jensen, T., Pichardie, D.: Proof-carrying code from certified abstract interpretation and fixpoint compression. Theor. Comput. Sci. 364(3), 273–291 (2006)

[BN98] Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1998)

[CDG$^+$02] Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree automata techniques and applications (2002), http://www.grappa.univ-lille3.fr/tata/

[CJPR05] Cachera, D., Jensen, T., Pichardie, P., Rusu, V.: Extracting a data flow analyser in constructive logic. Theor. Comput. Sci. 342(1), 56–78 (2005)

[FGVTT04] Feuillade, G., Genet, T., Viet Triem Tong, V.: Reachability Analysis over Term Rewriting Systems. JAR 33(3-4), 341–383 (2004)

[Gen97] Genet, T.: Decidable approximations of sets of descendants and sets of normal forms (extended version). Technical Report RR-3325, INRIA (1997)

[Gen98] Genet, T.: Decidable approximations of sets of descendants and sets of normal forms. In: Nipkow, T. (ed.) RTA 1998. LNCS, vol. 1379, pp. 151–165. Springer, Heidelberg (1998)

[GK00] Genet, T., Klay, F.: Rewriting for Cryptographic Protocol Verification. In: McAllester, D. (ed.) CADE 2000. LNCS, vol. 1831. Springer, Heidelberg (2000)

[GT95] Gilleron, R., Tison, S.: Regular tree languages and rewrite systems. Fundamenta Informaticae 24, 157–175 (1995)

[GTTVTT03] Genet, T., Tang-Talpin, Y.-M., Viet Triem Tong, V.: Verification of Copy Protection Cryptographic Protocol using Approximations of Term Rewriting Systems. In: WITS 2003 (2003)

[GVTT00] Genet, T., Viet Triem Tong, V.: Timbuk 2.0 – a Tree Automata Library. IRISA / Université de Rennes 1 (2000), http://www.irisa.fr/lande/genet/timbuk/

[Jac96] Jacquemard, F.: Decidable approximations of term rewriting systems. In: Ganzinger, H. (ed.) Proc. 7th RTA Conf., New Brunswick (New Jersey, USA), pp. 362–376. Springer, Heidelberg (1996)

[KN03] Klein, G., Nipkow, T.: Verified bytecode verifiers. TCS 298 (2003)

[LT00] Letouzey, P., Théry, L.: Formalizing stalmarck's algorithm in coq. In: Aagaard, M.D., Harrison, J. (eds.) TPHOLs 2000. LNCS, vol. 1869. Springer, Heidelberg (2000)

[Ret99] Rety, P.: Regular Sets of Descendants for Constructor-based Rewrite Systems. In: Ganzinger, H., McAllester, D., Voronkov, A. (eds.) LPAR 1999. LNCS, vol. 1705. Springer. Heidelberg (1999)

[RGL01] Rival, X., Goubault-Larrecq, J.: Experiments with finite tree automata in coq. In: Proc. of TPHOL 2001. LNCS. Springer, Heidelberg (2001)

[Tak04] Takai, T.: A Verification Technique Using Term Rewriting Systems and Abstract Interpretation. In: van Oostrom, V. (ed.) RTA 2004. LNCS, vol. 3091, pp. 119–133. Springer, Heidelberg (2004)

[ZD06] Zunino, R., Degano, P.: Handling exp, $\times$ (and timestamps) in protocol analysis. In: Aceto, L., Ingólfsdóttir, A. (eds.) FOSSACS 2006 and ETAPS 2006. LNCS, vol. 3921, pp. 413–427. Springer, Heidelberg (2006)

Automated Induction with Constrained Tree Automata[⋆,⋆⋆]

Adel Bouhoula[1] and Florent Jacquemard[2]

[1] Higher School of Communications of Tunis (Sup'Com), University of November 7th at Carthage, Tunisia
adel.bouhoula@supcom.rnu.tn
[2] INRIA and LSV, CNRS/ENS Cachan, France
florent.jacquemard@inria.fr

Abstract. We propose a procedure for automated implicit inductive theorem proving for equational specifications made of rewrite rules with conditions and constraints. The constraints are interpreted over constructor terms (representing data values), and may express syntactic equality, disequality, ordering and also membership in a fixed tree language. Constrained equational axioms between constructor terms are supported and can be used in order to specify complex data structures like sets, sorted lists, trees, powerlists...

Our procedure is based on tree grammars with constraints, a formalism which can describe exactly the initial model of the given specification (when it is sufficiently complete and terminating). They are used in the inductive proofs first as an induction scheme for the generation of subgoals at induction steps, second for checking validity and redundancy criteria by reduction to an emptiness problem, and third for defining and solving membership constraints.

We show that the procedure is sound and refutationally complete. It generalizes former test set induction techniques and yields natural proofs for several non-trivial examples presented in the paper, these examples are difficult (if not impossible) to specify and carry on automatically with other induction procedures.

1 Introduction

Given a specification $\mathcal{R}$ of a program or system S made of equational Horn clauses, proving a property P for S generally amounts to show the validity of P in the minimal Herbrand model of $\mathcal{R}$, also called *initial model* of $\mathcal{R}$ (inductive validity). In this perspective, it is important to have automated induction theorem proving procedures supporting a specification language expressive enough to axiomatize complex data structures like sets, sorted lists, powerlists, complete binary trees, *etc.* Moreover, it is also important to be able to automatically generate induction schemas used for inductive proofs in order to minimize user interaction. However,

[⋆] A long version of this extended abstract is available as a research report [3].
[⋆⋆] This work has been partially supported by INRIA/DGRSRT grants 06/I09 and 0804.

theories of complex data structures generate complex induction schemes, and the automation of inductive proofs is therefore difficult for such theories.

It is common to assume that $\mathcal{R}$ is built with *constructor function symbols* (to construct terms representing data) and *defined symbols* (representing the operations defined on constructor terms). Assuming in addition the *sufficient completeness* of $\mathcal{R}$ (every ground (variable-free) term is reducible, using the axioms of $\mathcal{R}$, to a constructor term) and the strong termination of the axioms of $\mathcal{R}$, a set of representants for the initial model of $\mathcal{R}$ (the model in which we want to proof the validity of conjectures) is the set of ground constructor terms not reducible by $\mathcal{R}_{\mathcal{C}}$ (the subset of equations of $\mathcal{R}$ between terms made of constructor symbols), called constructor *normal forms*.

In the case where the constructors are *free* ($\mathcal{R}_{\mathcal{C}} = \emptyset$), the set of constructor normal forms is simply the set of ground terms built with constructors and it is very easy to define an induction schema. This situation is therefore convenient for inductive reasoning, and many inductive theorem provers require free constructors, termination and sufficient completeness. However, it is not expressive enough to define complex data structures. With rewrite rules between constructors, the definition of induction schema is more complex, and requires a finite description of the set of constructor normal-forms. Some progress has been done *e.g.* in [4] and [5] in the direction of handling specification with non-free constructors, with severe restrictions (see related work below).

Tree automata (TA), or equivalently regular tree grammars, permit a finite representation of the set of constructor normal-forms when $\mathcal{R}_{\mathcal{C}}$ is a left-linear rewrite system (set of rewrite rules without multiple occurrences of variables in their left-hand-sides). Indeed, on one hand TA can do linear pattern-matching, hence they can recognize terms which are reducible by $\mathcal{R}_{\mathcal{C}}$, and on the other hand, the class of TA languages is closed under complementation. When the axioms of $\mathcal{R}_{\mathcal{C}}$ are not linear, or are constrained, some extensions of TA (or grammars) are necessary, with transitions able to check constraints on the term in input, see e.g. [8].

In this paper, we propose a framework for inductive theorem proving for theories containing constrained rewrite rules between constructor terms and conditional and constrained rewrite rules for defined functions. The key idea is a strong and natural integration of tree grammars with constraints in an implicit induction procedure, where they are used as induction schema. Very roughly, our procedure starts with the automatic computation of an induction schema, in the form of a constrained tree grammar generating constructor normal form. This grammar is used later for the generation of subgoals from a conjecture C, by instantiation of variables using the grammar's production rules, triggering induction steps during the proof. All generated subgoals are either deleted, following some criteria, or they are reduced, using axioms or induction hypotheses, or conjectures not yet proved, providing that they are smaller than the goal to be proved. Reduced subgoals become then new conjectures and C becomes an induction hypothesis. Moreover, constrained tree grammars are used as a decision procedure for checking the deletion criteria during induction steps.

Our method subsumes former test set induction procedures like [6,1,4], by reusing former theoretical works on tree automata with constraints. It is *sound* and *refutationally complete* (any conjecture that is not valid in the initial model will be disproved) when $\mathcal{R}$ is sufficiently complete and the constructor subsystem $\mathcal{R}_{\mathcal{C}}$ is terminating. Without the above hypotheses, it still remains sound and refutationally complete for a restricted kind of conjectures, where all the variables are constrained to belong to the language of constructor normal forms. This restriction is expressible in the specification language (see below). When the procedure fails, it implies that the conjecture is not an inductive theorem, provided that $\mathcal{R}$ is *strongly* complete (a stronger condition for sufficient completeness) and ground confluent. There is no requirement for *termination* of the whole set of rules $\mathcal{R}$, unlike [6,1], but instead only for separate termination of the respective sets of rules for defined function and for the constructors.

Moreover, if a conjecture C restricted as above is proved in a sufficiently complete specification $\mathcal{R}$ and $\mathcal{R}$ is further consistently extended into $\mathcal{R}'$ with additional axioms for specifying *partial* (non-constructor) functions, then the former proof of C remains valid in $\mathcal{R}'$, see Section 5.

The support of constraints permits in some cases to use the constrained completion technique of [16] in order to transform a non-terminating theory into a terminating one, by the addition of ordering constraints in constructor rules, see Section 4.5. It permits in particular to make proofs modulo non orientable axioms, without having to modify the core of our procedure.

We shall consider a specification of ordered lists as a running example throughout the paper. Consider first non-stuttering lists (lists which do not contain two equal successive elements) built with the constructor symbols $\emptyset$ (empty list) and *ins* (list insertion) and following this rewrite rule:

$$ins(x, ins(x, y)) \to ins(x, y) \qquad (\mathsf{c}_0)$$

Rewrite rules can be enriched with constraints built on predicates with a fixed interpretation on ground constructor terms. For example, using ordering constraints built with $\succ$ we can specify ordered lists by the following axiom:

$$ins(x_1, ins(x_2, y)) \to ins(x_2, ins(x_1, y)) \, [\![x_1 \succ x_2]\!] \qquad (\mathsf{c}_1)$$

Another interesting example is the case of membership constraints of the form $x : L$ where L is a fixed regular tree language (containing only terms made of constructor symbols). We consider also stronger constraints which restrict constructor terms to be in normal form (i.e. not reducible by the axioms). Let us come back to the example of non-stuttering sorted lists (sorted lists without duplication), and add to the above rules the axioms below which define a membership predicate $\Subset$, using the information that lists are sorted:

$$\begin{aligned} x \Subset \emptyset &\to \mathsf{false} && (\mathsf{m}'_0)\\ x_1 \Subset ins(x_2, y_2) &\to \mathsf{true} \, [\![x_1 \approx x_2]\!] && (\mathsf{m}'_1)\\ x_1 \Subset y_1 &\to \mathsf{false} \, [\![y_1 \approx ins(x_2, y_2), x_1 \prec x_2, y_1{:}\mathsf{NF}]\!] && (\mathsf{m}'_2)\\ x_1 \Subset ins(x_2, y_2) &\to x_1 \Subset y_2 \, [\![x_2 \prec x_1]\!] && (\mathsf{m}'_3) \end{aligned}$$

The constraint y_1:NF expresses the fact that this subterm is a constructor term in normal form, i.e. that it is a sorted list. Without this constraint, the specification would be inconsistent. Indeed, let us consider the ground term $t = 0 \Subset ins(s(0), ins(0, \emptyset))$. This term t can be reduced into both true and false, since $ins(s(0), ins(0, \emptyset))$ is not in normal form. Using constraints of the form $. : \mathsf{NF}$ as above also permits the user to specify, directly in the rewrite rules, some ad-hoc reduction strategies for the application of rewriting. Such strategies include for instance several refinements of the innermost strategy which corresponds to the *call by value* computation in functional programming languages, where arguments are fully evaluated before the function application.

Related work. The principle of our procedure is close to test-set induction approaches [6,1]. The real novelty here is that test-sets are replaced by constrained tree grammars, the latter being more precise induction schemes. Indeed, they provide an *exact* finite description of the initial model of the given specification, (under some assumptions like sufficient completeness and termination for axioms), whereas cover-sets and test-sets are over-approximative in similar cases.

The first author and Jouannaud [4] have used tree automata techniques to generalize test set induction to specifications with non-free constructors. This work has been generalized in [5] for membership equational logic. These approaches, unlike the procedure presented in this paper, work by transforming the initial specification in order to get rid of rewrite rules for constructors. Moreover, the axioms for constructors are assumed to be unconstrained and unconditional *left-linear* rewrite rules, which is still too restrictive for the specification of structures like sets or sorted lists...

Kapur [15] has proposed a method (implemented in the system RRL) for mechanizing cover set induction if the constructors are not free. This handles in particular the specification of powerlists or sorted lists. We show in Section 5 how our method can address similar problems.

We describe in [3] two proofs, done resp. by Jared Davis and Sorin Stratulat, of a conjecture on sorted lists, done resp. by Jared Davis and Sorin Stratulat, with ACL2 using a library for ordered sets [12] and with SPIKE [6,1,17]. Both proofs require the addition of non-trivial lemmas whereas our procedure can prove the conjecture without additional lemma.

2 Preliminaries

The reader is assumed familiar with the basic notions of term rewriting [13] and first-order logic. Notions and notations not defined here are standard.

Terms and substitutions. We assume given a many sorted signature $(\mathcal{S}, \mathcal{F})$ (or simply $\mathcal{F}$, for short) where $\mathcal{S}$ is a set of *sorts* and $\mathcal{F}$ is a finite set of function symbols with arities. We assume moreover that the signature $\mathcal{F}$ comes in two parts, $\mathcal{F} = \mathcal{C} \uplus \mathcal{D}$ where $\mathcal{C}$ a set of *constructor symbols*, and $\mathcal{D}$ is a set of *defined symbols*. Let $\mathcal{X}$ be a family of sorted variables. We sometimes denote variables with sort exponent like x^S in order to indicate that x has sort $S \in \mathcal{S}$. The set of well-sorted terms over $\mathcal{F}$ (resp. constructor well-sorted terms) with variables in $\mathcal{X}$ will be denoted by $\mathcal{T}(\mathcal{F},\mathcal{X})$ (resp. $\mathcal{T}(\mathcal{C},\mathcal{X})$). The subset of $\mathcal{T}(\mathcal{F},\mathcal{X})$ (resp. $\mathcal{T}(\mathcal{C},\mathcal{X})$) of variable-free terms, or *ground* terms, is denoted $\mathcal{T}(\mathcal{F})$ (resp. $\mathcal{T}(\mathcal{C})$). We assume that each sort contains a ground term. The sort of a term $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$ is denoted $sort(t)$.

A term t is identified as usual with a function from its set of *positions* (strings of positive integers) $\mathcal{P}os(t)$ to symbols of $\mathcal{F}$ and $\mathcal{X}$, where positions are strings of positive integers. We denote the empty string (root position) by Λ. The length of a position p is denoted $|p|$. The *depth* of a term t, denoted $d(t)$, is the maximum of $\{|p| \mid p \in \mathcal{P}os(t)\}$. The *subterm* of t at position p is denoted by $t|_p$. The result of replacing $t|_p$ with s at position p in t is denoted by $t[s]_p$. This notation is also used to indicate that s is a subterm of t, in which case p may be omitted. We denote the set of variables occurring in t by $var(t)$. A term t is *linear* if every variable of $var(t)$ occurs exactly once in t.

A *substitution* is a finite mapping $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ where $x_1, \ldots, x_n \in \mathcal{X}$ and $t_1, \ldots t_n \in \mathcal{T}(\mathcal{F},\mathcal{X})$. As usual, we identify substitutions with their morphism extension to terms. A *variable renaming* is a substitution mapping variables to variables. We use postfix notation for substitutions application and composition. A substitution σ is *grounding* for a term t if $t\sigma$ is ground.

Constraints and constrained terms. We assume given a constraint language $\mathcal{L}$, which is a finite set of predicate symbols with a recursive Boolean interpretation in the domain of ground constructor terms of $\mathcal{T}(\mathcal{C})$. Typically, $\mathcal{L}$ may contain the syntactic equality $. \approx .$ (syntactic disequality $. \not\approx .$), some (recursive) simplification ordering $. \prec .$ on ground constructor terms (for instance a lexicographic path ordering [13]), and membership $. : L$ to a fixed tree language $L \subseteq \mathcal{T}(\mathcal{C})$ (like for instance the languages of well sorted terms or constructor terms in normal-form). *Constraints* on the language $\mathcal{L}$ are Boolean combinations of atoms of the form $P(t_1, \ldots, t_n)$ where $P \in \mathcal{L}$ and $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{C},\mathcal{X})$. By convention, an empty combination is interpreted to true.

The application of substitutions is extended from terms to constraints in a straightforward way, and we may therefore define a solution for a constraint c as a (constructor) substitution σ grounding for all terms in c and such that $c\sigma$ is interpreted to true. The set of solutions of the constraint c is denoted $sol(c)$. A constraint c is *satisfiable* if $sol(c) \neq \emptyset$ (and *unsatisfiable* otherwise).

A *constrained term* $t\,[\![c]\!]$ is a linear term $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$ together with a constraint c, which may share some variables with t. Note that the assumption that t is linear is not restrictive, since any non linearity may be expressed in the constraint, for instance $f(x,x)\,[\![c]\!]$ is semantically equivalent to $f(x,x')\,[\![c \wedge x \approx x']\!]$, where the variable x' does not occur in c.

Constrained clauses. A *literal* is an equation $s = t$ or a disequation $s \neq t$ or an oriented equation $s \to t$ between two terms. A *constrained clause* $C\,[\![c]\!]$ is a disjunction C of literals together with a constraint c. A constrained clause $C\,[\![c]\!]$ is said to *subsume* a constrained clause $C'\,[\![c']\!]$ if there is a substitution σ such that $C\sigma$ is a sub-clause of C' and $c' \wedge \neg c\sigma$ is unsatisfiable.

A *tautology* is a constrained clause $s_1 = t_1 \vee \ldots \vee s_n = t_n \llbracket d \rrbracket$ such that d is a conjunction of equational constraints, $d = u_1 \approx v_1 \wedge \ldots \wedge u_k \approx v_k$ and there exists $i \in [1..n]$ such that $s_i\sigma = t_i\sigma$ where σ is the mgu of d.

Constrained rewriting. A *conditional constrained rewrite rule* is a constrained clause of the form $\Gamma \Rightarrow l \to r \llbracket c \rrbracket$ such that Γ is a conjunction of equations, called the *condition* of the rule, the terms l and r (called resp. left- and right-hand side) are linear and have the same sort, and c is a constraint. When the condition Γ is empty, it is called a *constrained rewrite rule.* A set of conditional constrained, resp. constrained, rules is called a *conditional constrained* (resp. *constrained*) *rewrite system.* Let $\mathcal{R}$ be a conditional constrained rewrite system. The relation $s \llbracket d \rrbracket$ rewrites to $t \llbracket d \rrbracket$ by $\mathcal{R}$, denoted $s \llbracket d \rrbracket \xrightarrow[\mathcal{R}]{} t \llbracket d \rrbracket$, is defined recursively by the existence of a rule $\rho \equiv \Gamma \Rightarrow \ell \to r \llbracket c \rrbracket \in \mathcal{R}$, a position $p \in \mathcal{P}os(s)$, and a substitution σ such that $s|_p = \ell\sigma$, $t|_p = r\sigma$, $d\sigma \wedge \neg c\sigma$ is unsatisfiable, and $u\sigma \downarrow_{\mathcal{R}} v\sigma$ for all $u = v \in \Gamma$. The transitive and reflexive transitive closures, of $\xrightarrow[\mathcal{R}]{}$ are denoted $\xrightarrow[\mathcal{R}]{+}$ and $\xrightarrow[\mathcal{R}]{*}$, and $u \downarrow_{\mathcal{R}} v$ stands for $\exists w,\ u \xrightarrow[\mathcal{R}]{*} w \xleftarrow[\mathcal{R}]{*} v$.

Note the semantical difference between conditions and constraints in rewrite rules. The validity of the condition is defined wrt the system $\mathcal{R}$ whereas the interpretation of constraint is fixed and independent from $\mathcal{R}$.

A constrained term $s \llbracket c \rrbracket$ is *reducible* by $\mathcal{R}$ if there is some $t \llbracket c \rrbracket$ such that $s \llbracket c \rrbracket \xrightarrow[\mathcal{R}]{} t \llbracket c \rrbracket$. Otherwise $s \llbracket c \rrbracket$ is called *irreducible*, or an $\mathcal{R}$-normal form. A substitution σ is *irreducible* by $\mathcal{R}$ if its image contains only $\mathcal{R}$-normal forms. A constrained term $t \llbracket c \rrbracket$ is *ground reducible* (resp. *ground irreducible*) if $t\sigma$ is reducible (resp. irreducible) for every irreducible solution σ of c grounding for t.

The system $\mathcal{R}$ is *terminating* if there is no infinite sequence $t_1 \xrightarrow[\mathcal{R}]{} t_2 \xrightarrow[\mathcal{R}]{} \ldots$, $\mathcal{R}$ is *ground confluent* if for any ground terms $u, v, w \in \mathcal{T}(\mathcal{F})$, $v \xleftarrow[\mathcal{R}]{*} u \xrightarrow[\mathcal{R}]{*} w$, implies that $v \downarrow_{\mathcal{R}} w$, and $\mathcal{R}$ is *ground convergent* if $\mathcal{R}$ is both ground confluent and terminating. The *depth* of a non-empty set $\mathcal{R}$ of rules, denoted $d(\mathcal{R})$, is the maximum of the depths of the left-hand sides of rules in $\mathcal{R}$.

Constructor specifications. We assume from now on given a conditional constrained rewrite system $\mathcal{R}$. The subset of $\mathcal{R}$ containing only function symbols from $\mathcal{C}$ is denoted $\mathcal{R}_{\mathcal{C}}$ and $\mathcal{R} \setminus \mathcal{R}_{\mathcal{C}}$ is denoted $\mathcal{R}_{\mathcal{D}}$.

Inductive theorems. A clause C is a *deductive theorem* of $\mathcal{R}$ (denoted $\mathcal{R} \models C$) if it is valid in any model of $\mathcal{R}$. A clause C is an *inductive theorem* of $\mathcal{R}$ (denoted $\mathcal{R} \models_{\mathcal{I}nd} C$) iff for all for all substitution σ grounding for C, $\mathcal{R} \models C\sigma$.

We shall need below to generalize the definition of inductive theorems to constrained clauses as follows: a constrained clause $C \llbracket c \rrbracket$ is an inductive theorem of $\mathcal{R}$ (denoted $\mathcal{R} \models_{\mathcal{I}nd} C \llbracket c \rrbracket$) if for all substitutions $\sigma \in sol(c)$ grounding for C we have $\mathcal{R} \models C\sigma$.

Completeness. A function symbol $f \in \mathcal{D}$ is *sufficiently complete* wrt $\mathcal{R}$ iff for all $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{C})$, there exists t in $\mathcal{T}(\mathcal{C})$ such that $f(t_1, \ldots, t_n) \xrightarrow[\mathcal{R}]{+} t$. We say that the system $\mathcal{R}$ is sufficiently complete iff every defined operator $f \in \mathcal{D}$ is sufficiently complete wrt $\mathcal{R}$. Let $f \in \mathcal{D}$ be a function symbol and let:

$$\Big\{\Gamma_1 \Rightarrow f(t^1_1, \ldots, t^1_k) \to r_1 \llbracket c_1 \rrbracket, \ldots, \Gamma_n \Rightarrow f(t^n_1, \ldots, t^n_k) \to r_n \llbracket c_n \rrbracket\Big\}$$

be a maximal subset of rules of $\mathcal{R}_{\mathcal{D}}$ whose left-hand sides are identical up to variable renamings $\mu_1, \ldots, \mu_n$, *i.e.* $f(t_1^1, \ldots, t_k^1)\mu_1 = f(t_1^2, \ldots, t_k^2)\mu_2 = \ldots f(t_1^n, \ldots, t_k^n)\mu_n$. We say that f is *strongly complete* wrt $\mathcal{R}$ (see [1]) if f is sufficiently complete wrt $\mathcal{R}$ and $\mathcal{R} \models_{\mathcal{I}nd} \Gamma_1\mu_1 \llbracket c_1\mu_1 \rrbracket \vee \ldots \vee \Gamma_n\mu_n \llbracket c_n\mu_n \rrbracket$ for every subset of $\mathcal{R}$ as above. The system $\mathcal{R}$ is said strongly complete if every function symbol $f \in \mathcal{D}$ is strongly complete wrt $\mathcal{R}$.

3 Constrained Grammars

Constrained tree grammars have been introduced in [7], in the context of automated induction. The idea of using such formalism for induction theorem proving is also in *e.g.* [4,10], because it is known that they can generate the languages of normal-forms for arbitrary term rewriting systems.

We present in this section the definitions and results suited to our purpose.

Definition 1. *A* constrained grammar $\mathcal{G} = (Q, \Delta)$ *is given by: 1. a finite set Q of non-terminals of the form $\llcorner u \lrcorner$, where u is a linear term of $\mathcal{T}(\mathcal{F}, \mathcal{X})$, 2. a finite set Δ of production rules of the form $\llcorner v \lrcorner := f(\llcorner u_1 \lrcorner, \ldots, \llcorner u_n \lrcorner) \llbracket c \rrbracket$ where $f \in \mathcal{F}$, $\llcorner v \lrcorner$, $\llcorner u_1 \lrcorner, \ldots, \llcorner u_n \lrcorner \in Q$ (modulo variable renaming) and c is a constraint.*

The non-terminals are always considered modulo variable renaming. In particular, we assume *wlog* (for technical convenience) that the above term $f(u_1, \ldots, u_n)$ is linear and that $var(v) \cap var(f(u_1, \ldots, u_n)) = \emptyset$.

3.1 Term Generation

We associate to a given constrained grammar $\mathcal{G} = (Q, \Delta)$ a finite set of new unary predicates of constraint of the form $.\!:\!\llcorner u \lrcorner$, where $\llcorner u \lrcorner \in Q$ (modulo variable renaming). Constraints of the form $t\!:\!\llcorner u \lrcorner$ called *membership constraints* and their interpretation is given below. The production relation between constrained terms $\vdash_{\mathcal{G}}^{y}$ is defined by:

$$t[y] \llbracket y\!:\!\llcorner v \lrcorner \wedge d \rrbracket \vdash_{\mathcal{G}}^{y} t[f(y_1, \ldots, y_n)] \llbracket y_1\!:\!\llcorner u_1 \lrcorner \wedge \ldots \wedge y_n\!:\!\llcorner u_n \lrcorner \wedge c \wedge d\tau \rrbracket$$

if there exists $\llcorner v \lrcorner := f(\llcorner u_1 \lrcorner, \ldots, \llcorner u_n \lrcorner) \llbracket c \rrbracket \in \Delta$ such that $f(u_1, \ldots, u_n) = v\tau$, and $y_1,\ldots,y_n$ are fresh variables. The variable y, constrained to be in the language defined by the non-terminal $\llcorner v \lrcorner$ is replaced by $f(y_1, \ldots, y_n)$ where the variables $y_1, \ldots, y_n$ are constrained to the respective languages of non-terminals $\llcorner u_1 \lrcorner, \ldots, \llcorner u_n \lrcorner$. The union of the relations $\vdash_{\mathcal{G}}^{y}$ for all y is denoted $\vdash_{\mathcal{G}}$ and the reflexive transitive and transitive closures of the relation $\vdash_{\mathcal{G}}$ are respectively denoted by $\vdash_{\mathcal{G}}^{*}$ and $\vdash_{\mathcal{G}}^{+}$ ($\mathcal{G}$ may be omitted).

Definition 2. *The language $L(\mathcal{G}, \llcorner u \lrcorner)$ is the set of ground terms t* generated *by $\mathcal{G}$ from a non-terminal $\llcorner u \lrcorner$, i.e. such that $y \llbracket y\!:\!\llcorner u \lrcorner \rrbracket \vdash^{*} t \llbracket c \rrbracket$ where c is satisfiable.*

Given $Q' \subseteq Q$, we write $L(\mathcal{G}, Q') = \bigcup_{\llcorner u \lrcorner \in Q'} L(\mathcal{G}, \llcorner u \lrcorner)$ and $L(\mathcal{G}) = L(\mathcal{G}, Q)$. Given a constrained grammar $\mathcal{G} = (Q, \Delta)$, we can now define $sol(t\!:\!\llcorner u \lrcorner)$, where $\llcorner u \lrcorner \in Q$, as $\{\sigma \mid t\sigma \in L(\mathcal{G}, \llcorner u \lrcorner)\}$.

Example 1. Let us consider the sort Nat of natural integers built with the constructor symbols 0 and s. These terms are generated by the grammar with production rules $\llcorner x^{\mathsf{Nat}}\lrcorner := 0$ and $\llcorner x^{\mathsf{Nat}}\lrcorner := s(\llcorner x_2{}^{\mathsf{Nat}}\lrcorner)$. $\diamond$

3.2 Normal Forms

In [3], we present the automatic construction of a constrained grammar $\mathcal{G}_{\mathrm{NF}}(\mathcal{R}_{\mathcal{C}}) = (Q_{\mathrm{NF}}(\mathcal{R}_{\mathcal{C}}), \Delta_{\mathrm{NF}}(\mathcal{R}_{\mathcal{C}}))$ which generates the language of ground $\mathcal{R}_{\mathcal{C}}$-normal forms. Its construction is a generalization of the one of [9]. Intuitively, it corresponds to the complementation and completion of a grammar for $\mathcal{R}_{\mathcal{C}}$-reducible terms (such a grammar does mainly pattern matching of left members of rewrite rules), where every subset of states (for the complementation) is represented by the most general common instance of its elements (if they are unifiable). Due to space limitations, we cannot describe the general construction of $\mathcal{G}_{\mathrm{NF}}(\mathcal{R}_{\mathcal{C}})$ here but we rather present the case where $\mathcal{R}_{\mathcal{C}}$ contains the constructor axioms given in introduction.

Example 2. Let $\mathcal{R}_{\mathcal{C}}$ contain the axioms (c_0) and (c_1) given in introduction. Let $\mathcal{G}_{\mathrm{NF}}(\mathcal{R}_{\mathcal{C}})$ contain the two productions rules given Example 1 and $\llcorner x^{\mathsf{Set}}\lrcorner := \emptyset$, $\llcorner ins(x, y)\lrcorner := ins(\llcorner x^{\mathsf{Nat}}\lrcorner, \llcorner x^{\mathsf{Set}}\lrcorner)$ (for singleton lists) and $\llcorner ins(x, y)\lrcorner := ins(\llcorner x^{\mathsf{Nat}}\lrcorner, \llcorner ins(x_2, y_2)\lrcorner)\, [\![x^{\mathsf{Nat}} \prec x_2]\!]$. Note that the variables in the non terminal $\llcorner ins(x_2, y_2)\lrcorner$ in the right member of the latter production rule have been renamed in order to be distinguished from the variables in the non terminal in the left member. This grammar $\mathcal{G}_{\mathrm{NF}}(\mathcal{R}_{\mathcal{C}})$ generates the set of ground constructor terms in normal-form for $\mathcal{R}_{\mathcal{C}}$. They represent the ordered lists of natural numbers (of sort List). $\diamond$

4 Inference System

In this section, we present an inference system for our inductive theorem proving procedure. The principle is the following: given a goal (conjecture) C, we use the grammar $\mathcal{G}_{\mathrm{NF}}(\mathcal{R}_{\mathcal{C}})$ of Section 3.2 in order to expand C into some subgoals. All the generated subgoals must then either be deleted, following some criteria, or be reduced, using axioms or induction hypotheses, or conjectures not yet proved, providing that they are smaller than the goal to be proved. Reduced subgoals become then new conjectures and C becomes an induction hypothesis.

The deletion criteria include tautologies, forward subsumption, clauses with an unsatisfiable constraint, and constructor clause that can be detected as inductively valid, under some conditions defined precisely below. The decision of these criteria, using $\mathcal{G}_{\mathrm{NF}}(\mathcal{R}_{\mathcal{C}})$, is discussed in Section 4.6.

The reduction of subgoals is performed with the rules defined in Sections 4.1 and 4.2. If a subgoal generated cannot be deleted or reduced, then the procedure stop with a refutation (the initial goal is not an inductive theorem of $\mathcal{R}$). If every subgoal is deleted, then the initial goal is an inductive theorem of $\mathcal{R}$.

The procedure may not terminate (the conditions in inference rules other than the deletion criteria are recursive calls of the procedure of the form $\mathcal{R} \models_{\mathcal{I}nd}$

subgoal). In this case appropriate lemmas should be added by the user in order to achieve termination.

4.1 Simplification Rules for Defined Functions

Our procedure uses the simplification rules for defined symbols presented in Figure 1. They simplify constrained clauses according to $\mathcal{R}_\mathcal{D}$ and to a set $\mathcal{H}$ of induction hypotheses (constrained clauses), which is given as the second component of the left-hand sides of rules. Inductive Rewriting simplifies goals using the axioms as well as instances of the induction hypotheses of $\mathcal{H}$, provided that they are smaller than the goal. The underlying induction principle is based on a well-founded ordering $\gg$ on constrained clauses (see [3]). This approach is more general than structural induction which is more restrictive concerning simplification with induction hypotheses (see e.g. [6]). Inductive Contextual Rewriting can be viewed as a generalization of a rule in [18] to handle constraints by recursively discharging them as inductive conjectures. Rewrite Splitting simplifies a clause which contains a subterm matching some left member of rule of $\mathcal{R}_\mathcal{D}$. This inference checks moreover that all cases are covered for the application of $\mathcal{R}_\mathcal{D}$, *i.e.* that for each ground substitution τ, the conditions and the constraints of at least one rule is true wrt τ. Note that this condition is always true when $\mathcal{R}$ is sufficiently complete, and hence that this check is superfluous in this case. Inductive Deletion deletes tautologies and clauses with unsatisfiable constraints.

Inductive Rewriting: $(\{C\,[\![c]\!]\}, \mathcal{H}) \rightarrow_\mathcal{D} \{C'\,[\![c]\!]\}$
if $\quad C\,[\![c]\!] \xrightarrow[\rho,\sigma]{} C'\,[\![c]\!]$, $l\sigma > r\sigma$ and $l\sigma > \Gamma\sigma$
where $\rho = \Gamma \Rightarrow l \rightarrow r\,[\![c]\!] \in \mathcal{R}_\mathcal{D} \cup \{\psi \mid \psi \in \mathcal{H}$ and $C\,[\![c]\!] \gg \psi\}$

Inductive Contextual Rewriting: $(\{\Upsilon \Rightarrow C[l\sigma]\,[\![c]\!]\}, \mathcal{H}) \rightarrow_\mathcal{D} \{\Upsilon \Rightarrow C[r\sigma]\,[\![c]\!]\}$
if $\mathcal{R} \models_{\mathcal{I}nd} \Upsilon \Rightarrow \Gamma\sigma\,[\![c \wedge c'\sigma]\!]$, $l\sigma > r\sigma$ and $\{l\sigma\} >^{mul} \Gamma\sigma$, where $\Gamma \Rightarrow l \rightarrow r\,[\![c']\!] \in \mathcal{R}_\mathcal{D}$

Rewrite Splitting: $(\{C[t]_p\,[\![c]\!]\}, \mathcal{H}) \rightarrow_\mathcal{D} \{\Gamma_i\sigma_i \Rightarrow C[r_i\sigma_i]_p\,[\![c \wedge c_i\sigma_i]\!]\}_{i\in[1..n]}$
if $\mathcal{R} \models_{\mathcal{I}nd} \Gamma_1\sigma_1\,[\![c_1\sigma_1]\!] \vee \ldots \vee \Gamma_n\sigma_n\,[\![c_n\sigma_n]\!]$, $t > r_i\sigma_i$ and $\{t\} >^{mul} \Gamma_i\sigma_i$, where the $\Gamma_i\sigma_i \Rightarrow l_i\sigma_i \rightarrow r_i\sigma_i\,[\![c_i\sigma_i]\!]$, $i \leq n$, are all the instances of rules in $\mathcal{R}_\mathcal{D}$ such that $l_i\sigma_i = t$

Inductive Deletion: $(\{C\,[\![c]\!]\}, \mathcal{H}) \rightarrow_\mathcal{D} \emptyset$ if $C\,[\![c]\!]$ is a tautology or c is unsatisfiable

Fig. 1. Simplification Rules for Defined Functions

4.2 Simplification Rules for Constructors

The simplification rules for constructors are presented in Figure 2. Rewriting simplifies goals with axioms from $\mathcal{R}_\mathcal{C}$. Partial Splitting eliminates ground reducible terms in a constrained clause $C\,[\![c]\!]$ by adding to $C\,[\![c]\!]$ the negation of constraint of some rules of $\mathcal{R}_\mathcal{C}$. Therefore, the saturated application of Partial splitting and Rewriting will always lead to Deletion or to ground irreducible constructor clauses. Finally, Deletion and Validity remove respectively tautologies and clauses with unsatisfiable constraints, and ground irreducible constructor theorems of $\mathcal{R}$.

Rewriting: $\{C\,[\![c]\!]\} \to_{\mathcal{C}} \{C'\,[\![c]\!]\}$ if $C\,[\![c]\!] \xrightarrow[\mathcal{R}_{\mathcal{C}}]{+} C'\,[\![c]\!]$ and $C\,[\![c]\!] \gg C'\,[\![c]\!]$

Partial Splitting: $\{C[l\sigma]_p\,[\![c]\!]\} \to_{\mathcal{C}} \{C[r\sigma]_p\,[\![c \wedge c'\sigma]\!], C[l\sigma]_p\,[\![c \wedge \neg c'\sigma]\!]\}$
if $l \to r\,[\![c']\!] \in \mathcal{R}_{\mathcal{C}}$, $l\sigma > r\sigma$, and neither $c'\sigma$ nor $\neg c'\sigma$ is a subformula of c

Deletion: $\{C\,[\![c]\!]\} \to_{\mathcal{C}} \emptyset$ if $C\,[\![c]\!]$ is a tautology or c is unsatisfiable

Validity: $\{C\,[\![c]\!]\} \to_{\mathcal{C}} \emptyset$ if $C\,[\![c]\!]$ is a ground irreducible constructor clause and $\mathcal{R} \models_{\mathcal{I}nd} C\,[\![c]\!]$

Fig. 2. Simplification Rules for Constructors

Simplification: $\dfrac{(\mathcal{E} \cup \{C\,[\![c]\!]\}, \mathcal{H})}{(\mathcal{E} \cup \mathcal{E}', \mathcal{H})}$ if $\{C\,[\![c]\!]\} \to_{\mathcal{C}} \mathcal{E}'$

Inductive Simplification: $\dfrac{(\mathcal{E} \cup \{C\,[\![c]\!]\}, \mathcal{H})}{(\mathcal{E} \cup \mathcal{E}', \mathcal{H})}$ if $(\{C\,[\![c]\!]\}, \mathcal{E} \cup \mathcal{H}) \to_{\mathcal{D}} \mathcal{E}'$

Narrowing: $\dfrac{(\mathcal{E} \cup \{C\,[\![c]\!]\}, \mathcal{H})}{(\mathcal{E} \cup \mathcal{E}_1 \cup \ldots \cup \mathcal{E}_n, \mathcal{H} \cup \{C\,[\![c]\!]\})}$
if $\{C_i\,[\![c_i]\!]\} \to_{\mathcal{C}} \mathcal{E}_i$, where $\{C_1\,[\![c_1]\!], \ldots, C_n\,[\![c_n]\!]\}$ is the set of all clauses such that $C\,[\![c]\!] \vdash^* C_i\,[\![c_i]\!]$ and $d(C_i) - d(C) \leq d(\mathcal{R}) - 1$

Inductive Narrowing: $\dfrac{(\mathcal{E} \cup \{C\,[\![c]\!]\}, \mathcal{H})}{(\mathcal{E} \cup \mathcal{E}_1 \cup \ldots \cup \mathcal{E}_n, \mathcal{H} \cup \{C\,[\![c]\!]\})}$
if $(C_i\,[\![c_i]\!], \mathcal{E} \cup \mathcal{H} \cup \{C\,[\![c]\!]\}) \to_{\mathcal{D}} \mathcal{E}_i$, where $\{C_1\,[\![c_1]\!], \ldots, C_n\,[\![c_n]\!]\}$ is the set of all clauses such that $C\,[\![c]\!] \vdash^+ C_i\,[\![c_i]\!]$ and $d(C_i) - d(C) \leq d(\mathcal{R}) - 1$

Subsumption: $\dfrac{(\mathcal{E} \cup \{C\,[\![c]\!]\}, \mathcal{H})}{(\mathcal{E}, \mathcal{H})}$ if $C\,[\![c]\!]$ is subsumed by another clause of $\mathcal{R} \cup \mathcal{E} \cup \mathcal{H}$

Disproof: $\dfrac{(\mathcal{E} \cup \{C\,[\![c]\!]\}, \mathcal{H})}{(\bot, \mathcal{H})}$ if no other rule applies to the clause $C\,[\![c]\!]$

Fig. 3. Induction Inference Rules

4.3 Induction Inference Rules

The main inference system is displayed in Figure 3. Its rules apply to pairs $(\mathcal{E}, \mathcal{H})$ whose components are respectively the sets of current conjectures and of inductive hypotheses. Two inference rules below, Narrowing and Inductive Narrowing, use the grammar $\mathcal{G}_{\mathrm{NF}}(\mathcal{R}_{\mathcal{C}})$ for instantiating variables. In order to be able to apply these inferences, we shall initiate the process by adding to the conjectures one membership constraint for each variable.

Definition 3. *Let $C\,[\![c]\!]$ be a constrained clause such that c contains no membership constraint. The* decoration *of $C\,[\![c]\!]$, denoted $decorate(C\,[\![c]\!])$ is the set of clauses $C\,[\![c \wedge x_1{:}\llcorner u_1 \lrcorner \wedge \ldots \wedge x_n{:}\llcorner u_n \lrcorner]\!]$ where $\{x_1, \ldots, x_n\} = var(C)$, and for all $i \in [1..n]$, $\llcorner u_i \lrcorner \in Q_{\mathrm{NF}}(\mathcal{R}_{\mathcal{C}})$ and $sort(u_i) = sort(x_i)$.*

The definition of *decorate* is extended to set of constrained clauses as expected. A constrained clause $C\,[\![c]\!]$ is said *decorated* if $c = d \wedge x_1{:}\llcorner u_1 \lrcorner \wedge \ldots \wedge x_n{:}\llcorner u_n \lrcorner$ where $\{x_1, \ldots, x_n\} = var(C)$, and for all $i \in [1..n]$, $\llcorner u_i \lrcorner \in Q_{\mathrm{NF}}(\mathcal{R}_{\mathcal{C}})$, $sort(u_i) = sort(x_i)$, and d does not contain membership constraints.

Simplification, resp. Inductive Simplification, reduces conjectures according to the rules of Section 4.2, resp. 4.1. Inductive Narrowing generates new subgoals by application of the production rules of the constrained grammar $\mathcal{G}_{\mathrm{NF}}(\mathcal{R}_{\mathcal{C}})$ until the obtained clause is deep enough to cover left-hand side of rules of $\mathcal{R}_{\mathcal{D}}$. Each obtained clause must be simplified by one the rules of Figure 1 (otherwise, if one instance cannot be simplified, then the rule Inductive Narrowing cannot be applied). For sake of efficiency, the application can be restricted to so called *induction variables*, as defined in [1] while preserving all the results of the next section. Narrowing is similar and uses the rules of Figure 2 for simplification. This rule permits to eliminate the ground reducible constructor terms in a clause by simplifying their instances, while deriving conjectures considered as new subgoals. The criteria on depth is the same for Inductive Narrowing and Narrowing and is a bit rough, for sake of clarity of the inference rules. However, in practice, it can be replaced by a tighter condition (with, *e.g.*, a distinction between $\mathcal{R}_{\mathcal{C}}$ and $\mathcal{R}_{\mathcal{D}}$) while preserving the results of the next section. Subsumption deletes clauses redundant with axioms of $\mathcal{R}$, induction hypotheses of $\mathcal{H}$ and other conjectures not yet proved (in $\mathcal{E}$).

Example 3. Let us come back to the running example of sorted lists, with the constructor system $\mathcal{R}_{\mathcal{C}}$ containing (c_0) and (c_1) and the defined system $\mathcal{R}_{\mathcal{D}}$ containing the axioms (m'_0-m'_3) given in introduction[1] together with the following axioms defining a variant $\in$ for the membership:

$$x \in \emptyset \rightarrow \mathsf{false} \qquad (\mathsf{m}_0)$$

$$x_1 \in ins(x_2, y) \rightarrow \mathsf{true}\,[\![x_1 \approx x_2]\!] \qquad (\mathsf{m}_1)$$

$$x_1 \in ins(x_2, y) \rightarrow x_1 \in y\,[\![x_1 \not\approx x_2]\!] \qquad (\mathsf{m}_2)$$

We show, using our procedure, that the conjecture $x \Subset y = x \in y$ is an inductive theorem of $\mathcal{R}$, i.e. that the two variants $\Subset$ and $\in$ of membership are equivalent. The normal-form grammar $\mathcal{G}_{\mathrm{NF}}(\mathcal{R}_{\mathcal{C}})$ is described in example 2. The decoration of the conjecture with its non-terminal gives the two clauses: $x \Subset y = x \in y\,[\![x{:}\llcorner x^{\mathsf{Nat}} \lrcorner, y{:}\llcorner x^{\mathsf{Set}} \lrcorner]\!]$ and $x \Subset y = x \in y\,[\![x{:}\llcorner x^{\mathsf{Nat}} \lrcorner, y{:}\llcorner ins(x_1, y_1) \lrcorner]\!]$.

The application of the production rules of $\mathcal{G}_{\mathrm{NF}}(\mathcal{R}_{\mathcal{C}})$ to the first of these clauses (in Narrowing) gives: $x \Subset \emptyset = x \in \emptyset$ which is reduced, using (m'_0) and (m_0), to the tautology $\mathsf{false} = \mathsf{false}$. For the second clause, applying $\mathcal{G}_{\mathrm{NF}}(\mathcal{R}_{\mathcal{C}})$ returns:

$$x \Subset ins(x_1, \emptyset) = x \in ins(x_1, \emptyset)\,[\![x, x_1{:}\llcorner x^{\mathsf{Nat}} \lrcorner]\!] \qquad (1)$$

[1] In (m'_2), the constraints $y_1 \approx ins(x_2, y_2), y_1{:}\mathsf{NF}$ can be replaced by $y_1{:}\llcorner ins(x_2, y_2) \lrcorner$.

$$x \Subset ins(x_1, ins(x_2, \emptyset)) = x \in ins(x_1, ins(x_2, \emptyset))\, [\![x, x_1, x_2 \colon \llcorner x \lrcorner^{\mathsf{Nat}}, x_1 \prec x_2]\!] \quad (2)$$

$$x \Subset ins(x_1, ins(x_2, y_2)) = x \in ins(x_1, ins(x_2, y_2)) \\ [\![x, x_1, x_2 \colon \llcorner x \lrcorner^{\mathsf{Nat}}, y_2 \colon \llcorner ins(x_3, y_3) \lrcorner, x_1 \prec x_2, x_2 \prec x_3]\!] \quad (3)$$

The subgoals (1) and (2) can be simplified by Rewrite Splitting with (m'_1), (m'_2) and (m'_3) into clauses reduced into tautologies (see [3] for details).

The subgoal (3) is implified by Rewrite Splitting with (m'_1-m'_3) into 3 clauses. Let us consider the third one, obtained with (m'_3): $x \Subset ins(x_2, y_2) = x \in ins(x_1, ins(x_2, y_2))\, [\![x, x_1, x_2, x_3 \colon \llcorner x \lrcorner^{\mathsf{Nat}}, y_2 \colon \llcorner ins(x_3, y_3) \lrcorner, x_1 \prec x_2, x_2 \prec x_3, x_1 \prec x]\!]$. It is simplified by Inductive Rewriting with (m_2) into:
$x \Subset ins(x_2, y_2) = x \in ins(x_2, y_2)\, [\![x, x_2, x_3 \colon \llcorner x \lrcorner^{\mathsf{Nat}}, y_2 \colon \llcorner ins(x_3, y_3) \lrcorner, x_2 \prec x_3]\!]$.
At this point, we are allowed to use the conjecture $x \Subset y = x \in y$ as an induction hypothesis with Inductive Rewriting, it returns the tautology:

$$x \Subset ins(x_2, y_2) = x \Subset ins(x_2, y_2)\, [\![x, x_2, x_3 \colon \llcorner x \lrcorner^{\mathsf{Nat}}, y_2 \colon \llcorner ins(x_3, y_3) \lrcorner, x_2 \prec x_3]\!]$$

The ommited details in the proof of the conjecture can be found in [3]. Note that this proof does not require the manual addition of lemma. $\diamond$

4.4 Soundness and Completeness

We show now that our inference system is sound and refutationally complete. The proof of soundness is not straightforward. The main difficulty is to make sure that the exhaustve application of the rules preserve a counterexample when one exists. We will show more precisely that a *minimal* counterexample is preserved along a *fair* derivation.

A *derivation* is a sequence of inference steps generated by a pair of the form $(\mathcal{E}_0, \emptyset)$, using the inference rules in $\mathcal{I}$, written $(\mathcal{E}_0, \emptyset) \vdash_{\mathcal{I}} (\mathcal{E}_1, \mathcal{H}_1) \vdash_{\mathcal{I}} \ldots$ It is called *fair* if the set of persistent constrained clauses $(\cup_i \cap_{j \geq i} \mathcal{E}_j)$ is empty or equal to $\{\bot\}$. The derivation is said to be a *disproof* in the latter case, and a *success* in the former.

Finite success is obtained when the set of conjectures to be proved is exhausted. Infinite success is obtained when the procedure diverges, assuming fairness. When it happens, the clue is to guess some lemmas which are used to subsume or simplify the generated infinite family of subgoals, therefore stopping the divergence. This is possible in principle with our approach, since lemmas can be specified in the same way as axioms are.

Theorem 1 (Soundness of successful derivations). *Assume that $\mathcal{R}_\mathcal{C}$ is terminating and that $\mathcal{R}$ is sufficiently complete. Let $\mathcal{D}_0$ be a set of unconstrained clauses and let $\mathcal{E}_0 = decorate(\mathcal{D}_0)$. If there exists a successful derivation $(\mathcal{E}_0, \emptyset) \vdash_{\mathcal{I}} (\mathcal{E}_1, \mathcal{H}_1) \vdash_{\mathcal{I}} \cdots$ then $\mathcal{R} \models_{\mathcal{I}nd} \mathcal{D}_0$.*

Proof. (sketch, see [3] for a complete proof). The proof uses the fact that, under the hypotheses of Theorem 1, $\mathcal{R} \models_{\mathcal{I}nd} \mathcal{E}_0$ implies $\mathcal{R} \models_{\mathcal{I}nd} \mathcal{D}_0$.

Intuitively, the reason is that in order to show that $\mathcal{R} \models_{\mathcal{I}nd} \mathcal{D}_0$, it is sufficient to show that $\mathcal{R} \models \mathcal{D}_0\sigma$ for all substitutions σ whose images contain only ground constructor terms in normal form. Every ground σ can indeed be normalized into a substitution of this form because $\mathcal{R}_{\mathcal{C}}$ is terminating and $\mathcal{R}$ sufficiently complete. By definition of the decoration, the membership constraints and by construction of $\mathcal{G}_{\mathrm{NF}}(\mathcal{R}_{\mathcal{C}})$, this sufficient condition is a consequence of $\mathcal{R} \models_{\mathcal{I}nd} \mathcal{E}_0$.

We then show that $\mathcal{R} \models_{\mathcal{I}nd} \mathcal{E}_0$ by minimal counter-example. Assume that $\mathcal{R} \not\models_{\mathcal{I}nd} \mathcal{E}_0$ and let D_0 be a clause, minimal wrt $\gg$, in the set:

$$\big\{D\sigma \;\big|\; D\,[\![d]\!] \in \cup_i \mathcal{E}_i, \sigma \in sol(d) \text{ is constructor and irreducible and } \mathcal{R} \not\models D\sigma\big\}.$$

Let $C\,[\![c]\!]$ be a clause of $\cup_i \mathcal{E}_i$ minimal by subsumption ordering and $\theta \in sol(c)$, irreducible and constructor ground substitution, be such that $C\theta = D_0$. We show in [3] that whatever inference, other than Disproof, is applied to $C\,[\![c]\!]$, a contradiction is obtained, hence that the above derivation is not successful. □

Since there are only two kinds of fair derivations, we obtain as a corollary:

Corollary 1 (Refutational completeness). *Assume that $\mathcal{R}_{\mathcal{C}}$ is terminating and that $\mathcal{R}$ is sufficiently complete. Let $\mathcal{D}_0$ be a set of unconstrained clauses and let $\mathcal{E}_0 = decorate(\mathcal{D}_0)$. If $\mathcal{R} \not\models_{\mathcal{I}nd} \mathcal{E}_0$, then all fair derivations starting from $(\mathcal{E}_0, \emptyset)$ end up with $(\bot, \mathcal{H})$.*

When we assume that all the variables in goals are decorated (restricting the domain for this variables to ground constructor irreducible terms), the above hypotheses that $\mathcal{R}_{\mathcal{C}}$ is terminating and $\mathcal{R}$ is sufficiently complete can be dropped.

Theorem 2 (Soundness of successful derivations). *Let $\mathcal{E}_0$ be a set of decorated constrained clauses. If there exists a successful derivation $(\mathcal{E}_0, \emptyset) \vdash_{\mathcal{I}} (\mathcal{E}_1, \mathcal{H}_1) \vdash_{\mathcal{I}} \cdots$ then $\mathcal{R} \models_{\mathcal{I}nd} \mathcal{E}_0$.*

Proof. (sketch). We use the second part of the proof of Theorem 1 (which does not use the sufficient completeness of $\mathcal{R}$ and termination of $\mathcal{R}_{\mathcal{C}}$). With the hypothesis that the clauses of $\mathcal{E}_0$ are decorated, the fact given at the beginning of this proof is indeed no more needed ($\mathcal{D}_0 = \mathcal{E}_0$). The restriction to substitutions into ground constructor normal forms in order to show that $\mathcal{R} \not\models_{\mathcal{I}nd} \mathcal{E}_0$ is made explicit by the membership constraints in the decoration. □

Corollary 2 (Refutational completeness). *Let $\mathcal{E}_0$ be a set of decorated constrained clauses. If $\mathcal{R} \not\models_{\mathcal{I}nd} \mathcal{E}_0$, then all fair derivations starting from $(\mathcal{E}_0, \emptyset)$ end up with $(\bot, \mathcal{H})$.*

We shall see in Section 5 some example of applications of Theorem 2 and Corollary 2 to specifications which are not sufficiently complete.

Theorem 3 (Soundness of disproof). *Assume that $\mathcal{R}$ is strongly complete and ground confluent. If a derivation starting from $(\mathcal{E}_0, \emptyset)$ returns the pair $(\bot, \mathcal{H})$, then $\mathcal{R} \not\models_{\mathcal{I}nd} \mathcal{E}_0$.*

4.5 Handling Non-terminating Constructor Systems

Our procedure applies rules of $\mathcal{R}_\mathcal{C}$ and $\mathcal{R}_\mathcal{D}$ only when they reduce the terms wrt the given simplification ordering $>$. This is ensured when the rewrite relation induced by $\mathcal{R}_\mathcal{C}$ and $\mathcal{R}_\mathcal{D}$ is compatible with $>$, and hence that $\mathcal{R}_\mathcal{C}$ and $\mathcal{R}_\mathcal{D}$ are terminating (separately). Note that this is in contrast with other procedures like [16] where the termination of the whole system $\mathcal{R}$ is required.

If $\mathcal{R}_\mathcal{C}$ is non-terminating then one can apply a constrained completion technique [16] in order to generate an equivalent orientable theory (with ordering constraints). The theory obtained (if the completion succeeds) can then be handled by our approach.

Example 4. Consider this non-terminating system for sets: $\{ins(x, ins(x, y)) = ins(x, y), ins(x, ins(x', y)) = ins(x', ins(x, y))\}$. Applying the completion procedure we obtain the constrained rules (c_0) and (c_1). $\diamond$

4.6 Decision Procedures for Conditions in Inferences

Constrained tree grammars are involved in the inferences Narrowing and Inductive Narrowing in order to generate subgoals from goals, by instantiation using the productions rules. They are also the key for the decision procedures applied in order to check the conditions of constraint unsatisfiability (in rules for rewriting and Inductive Deletion, Deletion, Subsumption), ground irreducibility and validity of ground irreducible constructor clauses (in the rules Validity, hence Simplification, and Disproof). These conditions are decided by reduction into the decision problem of emptiness (of $L(\mathcal{G}, \llcorner u \lrcorner)$) for constrained tree grammars build from $\mathcal{G}_{\mathrm{NF}}(\mathcal{R}_\mathcal{C})$. The decision rely on similar decision results for constrained tree automata, some cases are summarized in [8]. The reductions are detailed in [3].

5 Handling Partial Specifications

The example of sorted lists (Example 3) can be treated with our procedure because it is based on a sufficiently complete and ground confluent conditional constrained TRS $\mathcal{R}$ whose constructor part $\mathcal{R}_\mathcal{C}$ is terminating. Indeed, under these hypotheses, Theorem 1 ensures the soundness of our procedure for proving inductive conjectures on this specification, and Corollary 1 and Theorem 3 ensure respectively refutational completeness and soundness of disproof.

For sound proofs of inductive theorems wrt specifications which are not sufficiently complete, we can rely on Theorem 2 and Corollary 2 which do not require sufficient completeness of the specification but instead suppose that the conjecture is decorated, i.e. that each of its variables is constrained to belong to a language associated to a non-terminal of the normal-form (constrained) grammar. In this section, we propose two applications of this principle of decoration of conjectures to the treatment of partial specifications.

Partially Defined Functions. An inductive proof of a decorated conjecture C in $\mathcal{R}$ remains valid in an extension of $\mathcal{R}$ (possibly not complete).

Theorem 4. *Assume that $\mathcal{R}$ is sufficiently complete and let $\mathcal{R}'$ be an consistent extension of $\mathcal{R}$ where $\mathcal{R_C}' = \mathcal{R_C}$ and $\mathcal{R_D}' = \mathcal{R_D} \cup \mathcal{R_D}''$ ($\mathcal{R_D}''$ defines additional partial defined functions). Let $\mathcal{E}_0$ be a set of decorated constrained clauses. Every derivation $(\mathcal{E}_0, \emptyset) \vdash_{\mathcal{I}} \cdots$ successful wrt $\mathcal{R}$ is also a successful derivation wrt $\mathcal{R}'$.*

In [3], we use Theorem 4 for the proof of conjectures on an extension of the specification of Example 3 with the incomplete definition of a function *min*.

Partial Constructors. The restriction to decorated conjectures also permits to deal with partial constructor functions. In this case, we are generally interested in proving conjectures only for constructor terms in the definition domain of the defined function (well-formed terms).

In [3], we present an example of automatic proof where $\mathcal{R_C}$ is such that the set of well-formed terms is the set of constructor $\mathcal{R_C}$-normal forms. Hence, decorating the conjecture with grammar's non-terminals, as in Theorem 2, amounts in this case at restricting the variables to be instantiated by well-formed terms.

The example is a specification of powerlists (lists of 2^n integers stored in the leaves of a complete binary tree) also treated in [15]. A particularity of this example is that $\mathcal{R_C}$ contains constraints of the form $t \sim t'$ meaning that t and t' are well-formed lists of the same length (*i.e.* balanced trees of the same depth). Such constraints are added to $\mathcal{G}_{\mathrm{NF}}(\mathcal{R_C})$ and we show that emptiness is decidable for these grammars by reduction to the same problem for visibly tree automata with one memory [11].

6 Conclusion

We have proposed a procedure for automated inductive theorem proving in specification made of conditional and constrained rewrite rules. Constraints in rules can serve to transform non terminating specifications into terminating ones (ordering constraints), define ad-hoc evaluation strategies (normal form constraints), or for the analysis of trace properties of infinite state systems like security protocols (constraints of membership in a regular tree language representing faulty traces [2]). The expressiveness and efficiency of the procedure are obtained by the use of constrained tree grammars as a finite representation of the initial model of specifications. They are used both as induction schema and for decision procedures.

The procedure presented in this paper is currently under implementation, on the top of the theorem prover SPASS, whose main loop is used for TA decision procedure by saturation, following the approach of e.g. [14].

Acknowledgments

We wish to thank Michael Rusinowitch, Hubert Comon-Lundh, Laurent Fribourg and Deepak Kapur for the fruitful discussions that we had together regarding this work. We are also grateful to Jared Davis and Sorin Stratulat for having processed the example on sorted lists with respectively ACL2 and SPIKE.

References

1. Bouhoula, A.: Automated theorem proving by test set induction. Journal of Symbolic Computation 23(1), 47–77 (1997)
2. Bouhoula, A., Jacquemard, F.: Verifying regular trace properties of security protocols with explicit destructors and implicit induction. In: Proc. of the workshop FCS-ARSPA, pp. 27–44 (2007)
3. Bouhoula, A., Jacquemard, F.: Automated induction with constrained tree automata. Research Report LSV-08-07, http://www.lsv.ens-cachan.fr/Publis
4. Bouhoula, A., Jouannaud, J.-P.: Automata-driven automated induction. Information and Computation 169(1), 1–22 (2001)
5. Bouhoula, A., Jouannaud, J.-P., Meseguer, J.: Specification and proof in membership equational logic. Theoretical Computer Science 236(1-2), 35–132 (2000)
6. Bouhoula, A., Rusinowitch, M.: Implicit induction in conditional theories. Journal of Automated Reasoning 14(2), 189–235 (1995)
7. Comon, H.: Unification et disunification. Théories et applications. PhD thesis, Institut Polytechnique de Grenoble (France) (1988)
8. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, C. Löding, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications (2007), http://www.grappa.univ-lille3.fr/tata
9. Comon, H., Jacquemard, F.: Ground reducibility is exptime-complete. Information and Computation 187(1), 123–153 (2003)
10. Comon-Lundh, H.: Handbook of Automated Reasoning, ch. 14. Elsevier, Amsterdam (2001)
11. Comon-Lundh, H., Jacquemard, F., Perrin, N.: Tree automata with memory, visibility and structural constraints. In: Seidl, H. (ed.) FOSSACS 2007. LNCS, vol. 4423, pp. 168–182. Springer, Heidelberg (2007)
12. Davis, J.: Finite set theory based on fully ordered lists. In: In 5th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2, 2004) Sets Library Website (2004), http://www.cs.utexas.edu/users/jared/osets/Web
13. Dershowitz, N., Jouannaud, J.-P.: Rewrite systems. In: Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics, pp. 243–320. MIT Press, Cambridge (1990)
14. Jacquemard, F., Rusinowitch, M., Vigneron, L.: Tree automata with equality constraints modulo equational theories. Journal of Logic and Algebraic Programming (to appear, 2008)
15. Kapur, D.: Constructors can be partial too. In: Essays in Honor of Larry Wos, MIT Press, Cambridge (1997)
16. Kirchner, C., Kirchner, H., Rusinowitch, M.: Deduction with symbolic constraints. Revue d'Intelligence Artificielle 4(3), 9–52 (1990); Special issue on Automatic Deduction
17. Stratulat, S.: A general framework to build contextual cover set induction provers. Journal of Symbolic Computation 32(4), 403–445 (2001)
18. Zhang, H.: Implementing contextual rewriting. In: Proc. 3rd Int. Workshop on Conditional Term Rewriting Systems (1992)

Author Index